Topics on CNN: Transfer Learning, Visualization, Neural Style, and Adversarial Examples

Yuan YAO

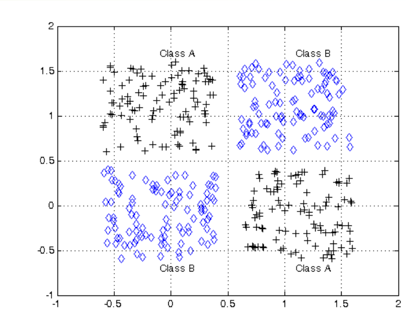
HKUST

Locality or Sparsity of Computation

Minsky and Papert, 1969

Perceptron can’t do **XOR** classification

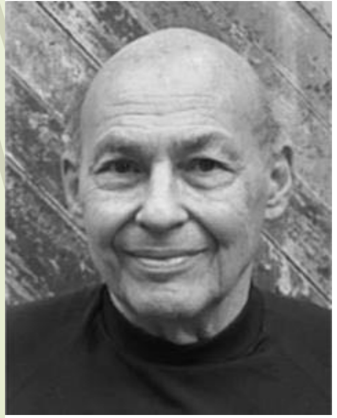
Perceptron needs infinite global information to compute **connectivity**



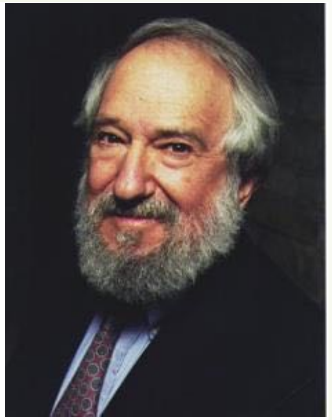
Locality or **Sparsity** is important:

Locality in time?

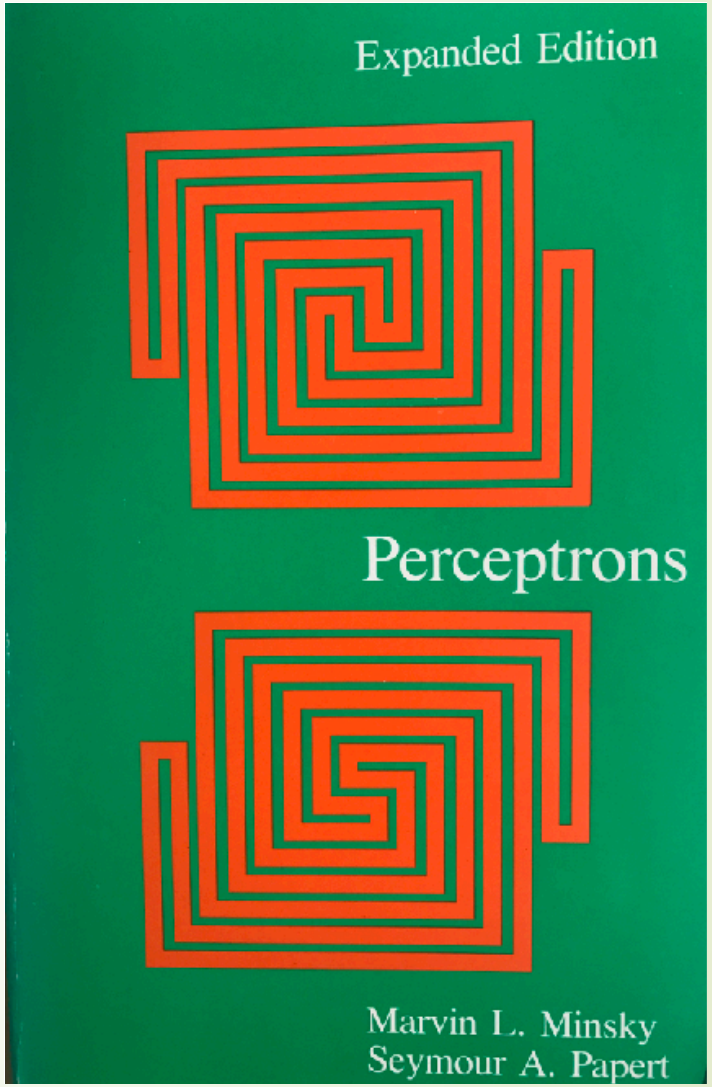
Locality in space?



Marvin Minsky

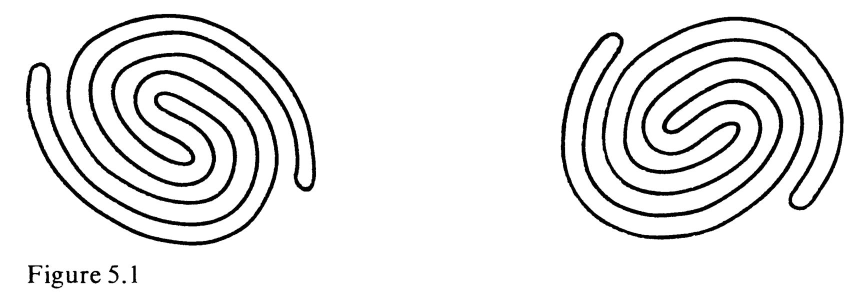


Seymour Papert



Connectivity is of infinite order

- Which one of these two figures is connected?



Theorem (Minsky-Papert'1969)

The decision function that $f(X) = [X \text{ is connected}]$ *for* $X \subseteq \mathbb{R}^p$ *is* **not of any finite order,** *i.e. for any* $k < \infty$*, there does not exist a (possibly of infinite members) family of* $\{\phi_\alpha(X) : \mathrm{supp}(\phi_\alpha) \leq k\}$ *whose supports are at most* k*, such that*

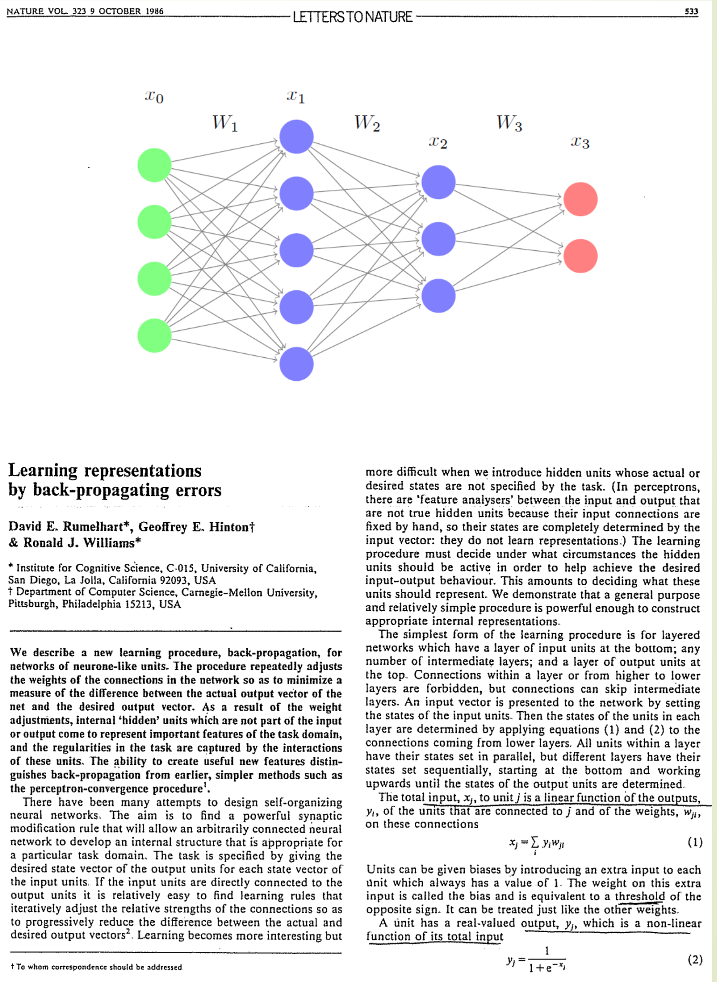
$$f(X) = \left[\sum_\alpha \phi_\alpha(X) \geq 0\right] \tag{21}$$

Multilayer Perceptrons (MLP) and Back-Propagation (BP) Algorithms

Rumelhart, Hinton, Williams (1986)
Learning representations by back-propagating errors, Nature, 323(9): 533-536

BP algorithms as **stochastic gradient descent** algorithms (**Robbins–Monro 1950; Kiefer-Wolfowitz 1951**) with Chain rules of Gradient maps

MLP classifies **XOR**, but the global hurdle on topology (connectivity) computation still exists



NATURE VOL. 323 9 OCTOBER 1986 — LETTERS TO NATURE — 533

Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

There have been many attempts to design self-organizing neural networks. The aim is to find a powerful synaptic modification rule that will allow an arbitrarily connected neural network to develop an internal structure that is appropriate for a particular task domain. The task is specified by giving the desired state vector of the output units for each state vector of the input units. If the input units are directly connected to the output units it is relatively easy to find learning rules that iteratively adjust the relative strengths of the connections so as to progressively reduce the difference between the actual and desired output vectors[2]. Learning becomes more interesting but more difficult when we introduce hidden units whose actual or desired states are not specified by the task. (In perceptrons, there are 'feature analysers' between the input and output that are not true hidden units because their input connections are fixed by hand, so their states are completely determined by the input vector: they do not learn representations.) The learning procedure must decide under what circumstances the hidden units should be active in order to help achieve the desired input–output behaviour. This amounts to deciding what these units should represent. We demonstrate that a general purpose and relatively simple procedure is powerful enough to construct appropriate internal representations.

The simplest form of the learning procedure is for layered networks which have a layer of input units at the bottom; any number of intermediate layers; and a layer of output units at the top. Connections within a layer or from higher to lower layers are forbidden, but connections can skip intermediate layers. An input vector is presented to the network by setting the states of the input units. Then the states of the units in each layer are determined by applying equations (1) and (2) to the connections coming from lower layers. All units within a layer have their states set in parallel, but different layers have their states set sequentially, starting at the bottom and working upwards until the states of the output units are determined.

The total input, x_j, to unit j is a linear function of the outputs, y_i, of the units that are connected to j and of the weights, w_{ji}, on these connections

$$x_j = \sum_i y_i w_{ji} \qquad (1)$$

Units can be given biases by introducing an extra input to each unit which always has a value of 1. The weight on this extra input is called the bias and is equivalent to a threshold of the opposite sign. It can be treated just like the other weights.

A unit has a real-valued output, y_j, which is a non-linear function of its total input

$$y_j = \frac{1}{1+e^{-x_j}} \qquad (2)$$

---

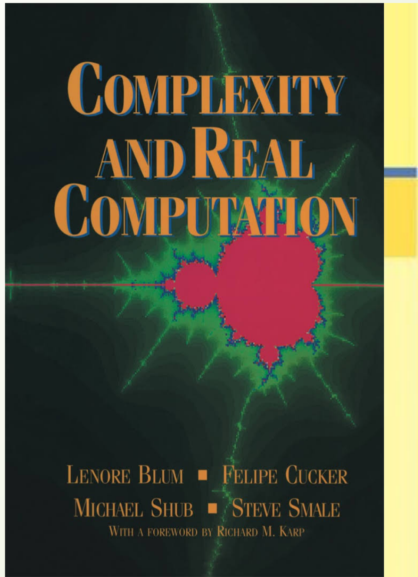
† To whom correspondence should be addressed.

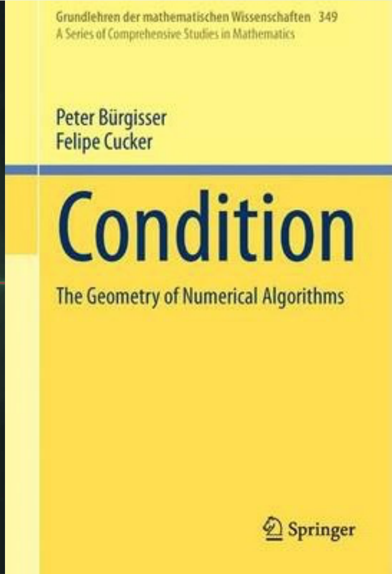
Topology can be learned with finite information if the manifold is *stable (finite condition number)*

Blum-Shub-Smale models of Real Computation

A Model of Real Computation

- Starting from **Blum, Shub, Smale** (1989)
- It admits inputs and operations (addition, substraction, multiplication, and (in the case of fields) division) of **real (complex) numbers** with *infinite precision*
- “The key importance of the **condition number**, which measures the closeness of a problem instance to the manifold of ill-posed instances, is clearly developed.” – Richard Karp





The Condition Number of a Manifold

Throughout our discussion, we associate to $\mathcal{M}$ a condition number $(1/\tau)$ where τ is defined as the largest number having the property: The open normal bundle about $\mathcal{M}$ of radius r is embedded in $\mathbb{R}^N$ for every $r < \tau$. Its image Tub_τ is a tubular neighborhood of $\mathcal{M}$ with its canonical projection map

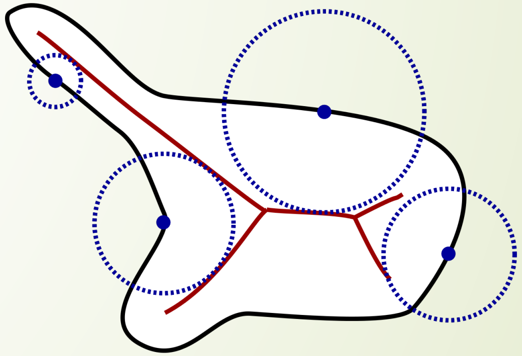
$$\pi_0 : \mathrm{Tub}_\tau \rightarrow \mathcal{M}.$$

Smallest Local Feature Size

$$G = \left\{x \in \mathbb{R}^N \text{ such that } \exists \text{ distinct } p, q \in \mathcal{M} \text{ where } d(x, \mathcal{M}) = \|x - p\| = \|x - q\|\right\},$$

where $d(x, \mathcal{M}) = \inf_{y \in \mathcal{M}} \|x - y\|$ is the distance of x to $\mathcal{M}$. The closure of G is called the medial axis and for any point $p \in \mathcal{M}$ the local feature size $\sigma(p)$ is the distance of p to the medial axis. Then it is easy to check that

$$\tau = \inf_{p \in \mathcal{M}} \sigma(p).$$



Find Homology with Finite Samples

[Niyogi, Smale, Weinberger (2008)]

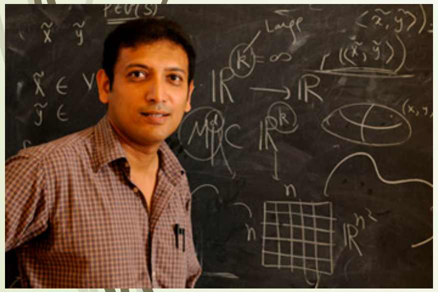
Theorem 3.1 *Let* $\mathcal{M}$ *be a compact submanifold of* $\mathbb{R}^N$ *with condition number* τ. *Let* $\bar{x} = \{x_1, \ldots, x_n\}$ *be a set of* n *points drawn in i.i.d. fashion according to the uniform probability measure on* $\mathcal{M}$. *Let* $0 < \epsilon < \tau/2$. *Let* $U = \bigcup_{x \in \bar{x}} B_\epsilon(x)$ *be a correspondingly random open subset of* $\mathbb{R}^N$. *Then for all*

$$n > \beta_1 \left(\log(\beta_2) + \log\left(\frac{1}{\delta}\right)\right),$$

the homology of U *equals the homology of* $\mathcal{M}$ *with high confidence (probability* $> 1 - \delta$).

$$\beta_1 = \frac{vol(\mathcal{M})}{(\cos^k(\theta_1))vol(B^k_{\epsilon/4})} \quad \text{and} \quad \beta_2 = \frac{vol(\mathcal{M})}{(\cos^k(\theta_2))vol(B^k_{\epsilon/8})}.$$

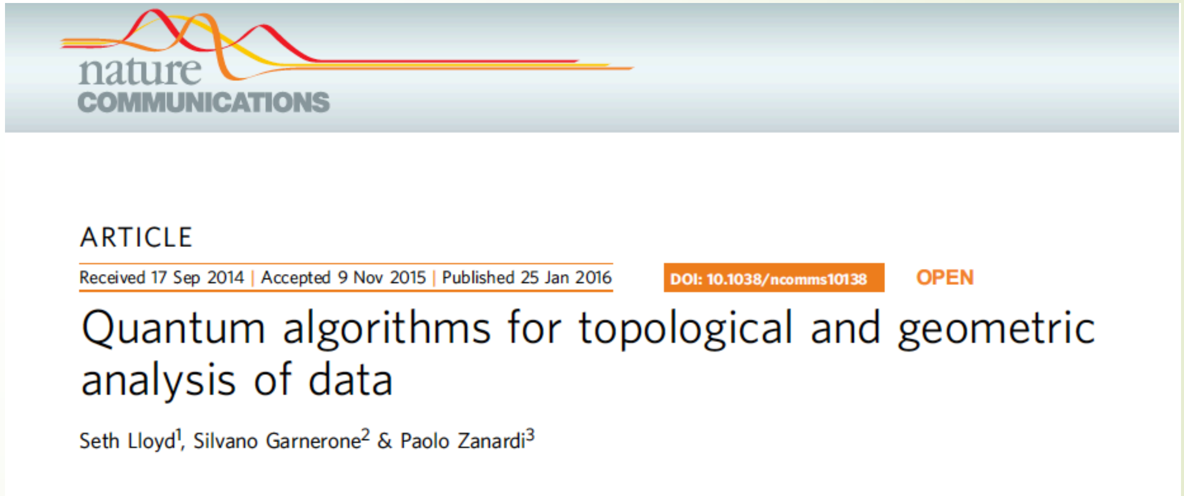
Here k *is the dimension of the manifold* $\mathcal{M}$ *and* $vol(B^k_\epsilon)$ *denotes the* k*-dimensional volume of the standard* k*-dimensional ball of radius* ϵ. *Finally,* $\theta_1 = \arcsin(\epsilon/8\tau)$ *and* $\theta_2 = \arcsin(\epsilon/16\tau)$.



Partha Niyogi@Chiccago, 1967-2010

Curse of Dimensionality and “Quantum Algorithms”

To construct a Rips-complex of dimension of n points: $O(2^n)$ number of simplices is needed in the worst case => O(poly(n)) in Quantum Algorithms



nature
COMMUNICATIONS

ARTICLE

Received 17 Sep 2014 | Accepted 9 Nov 2015 | Published 25 Jan 2016 DOI: 10.1038/ncomms10138 OPEN

Quantum algorithms for topological and geometric analysis of data

Seth Lloyd[1], Silvano Garnerone[2] & Paolo Zanardi[3]

A Proof of Concept Demonstration by 6-photon Quantum Computer [Huang et al. 2018, arXiv:1801.06316]

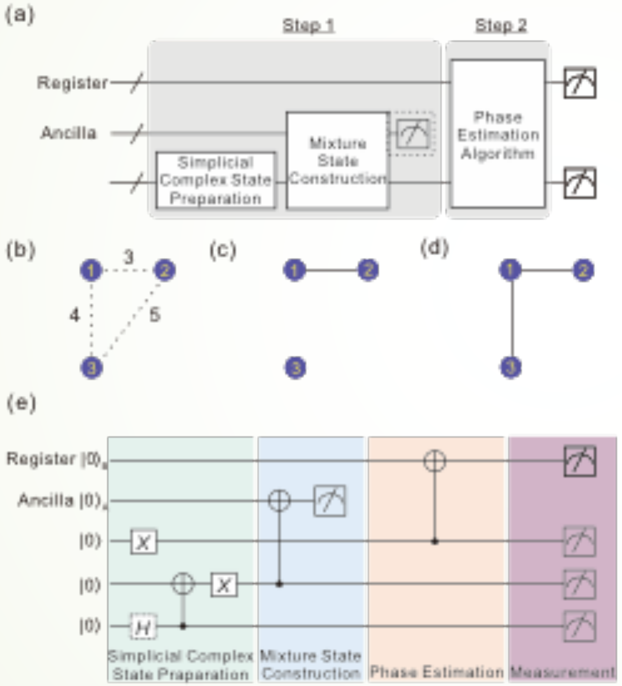


FIG. 2. Quantum circuit for quantum TDA. (a) Outline of the original quantum circuit. (b) A scatterplot including three data points. (c) Graph representation of the 1-simplices state $|\varphi\rangle_1^{\epsilon_1} = |110\rangle$ for $3 < \epsilon_1 < 4$. The first and second data points are connected by an edge. (d) Graph representation of 1-simplices state $|\varphi\rangle_1^{\epsilon_2} = (|110\rangle + |101\rangle)/\sqrt{2}$ for $4 < \epsilon_2 < 5$. The first data point is connected to the second and third points by two edges. (e) Optimized circuit with 5 qubits. The blocks with different colors represent the four basic stages.

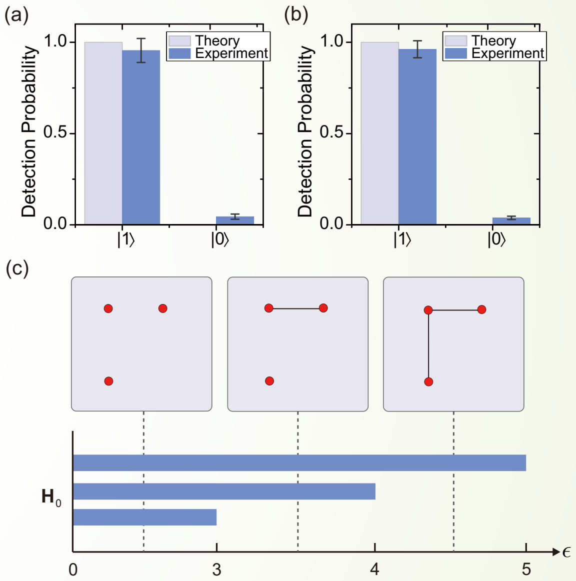
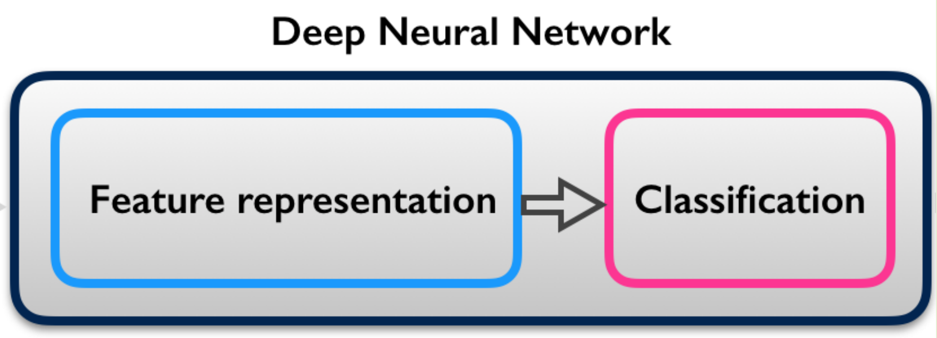


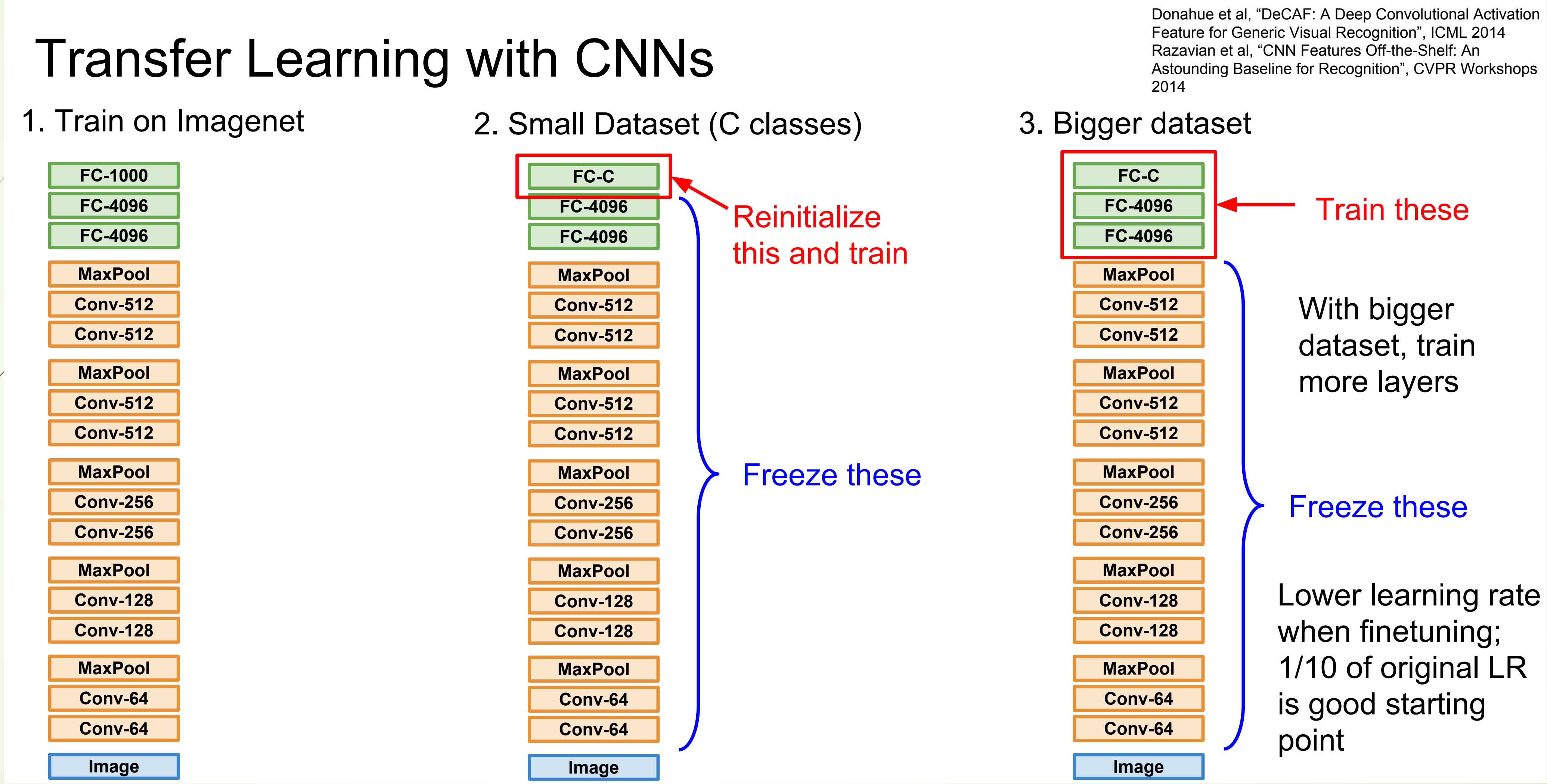
FIG. 4. Final experimental results. The output is determined by measuring the eigenvalue register in the Pauli-Z basis. Measured expectation values (blue bars) and theoretically predicted values (gray bars) are shown for two different 1-simplices state inputs: (a) $|\varphi\rangle_1^{\epsilon_1} = |110\rangle$, (b) $|\varphi\rangle_1^{\epsilon_2} = (|110\rangle + |101\rangle)/\sqrt{2}$. Error bars represent one standard deviation, deduced from propagated Poissonian counting statistics of the raw detection events. (c) The barcode for $0 < \epsilon < 5$. Since no k-dimensional holes for $k \geq 1$ exist at these scales, only the 0-th Betti barcode is given here. For $0 < \epsilon < 3$, there is no connection between each point, so the 0-th Betti number is equal to the number of points. That is, there are three bars at $0 < \epsilon < 3$. At scales of $3 < \epsilon_1 < 4$ and $4 < \epsilon_2 < 5$, the 0-th Betti number are 2 and 1.

Transfer Learning: Fine Tuning

Transfer Learning?



- Filters learned in first layers of a network are transferable from one task to another
- When solving another problem, no need to retrain the lower layers, just fine tune upper ones
- Is this simply due to the large amount of images in ImageNet?
- Does solving many classification problems simultaneously result in features that are more easily transferable?
- Does this imply filters can be learned in unsupervised manner?
- Can we characterize filters mathematically?



Transfer Learning with CNNs

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

1. Train on Imagenet

FC-1000
FC-4096
FC-4096
MaxPool
Conv-512
Conv-512
MaxPool
Conv-512
Conv-512
MaxPool
Conv-256
Conv-256
MaxPool
Conv-128
Conv-128
MaxPool
Conv-64
Conv-64
Image

2. Small Dataset (C classes)

FC-C
FC-4096
FC-4096
MaxPool
Conv-512
Conv-512
MaxPool
Conv-512
Conv-512
MaxPool
Conv-256
Conv-256
MaxPool
Conv-128
Conv-128
MaxPool
Conv-64
Conv-64
Image

Reinitialize this and train

Freeze these

3. Bigger dataset

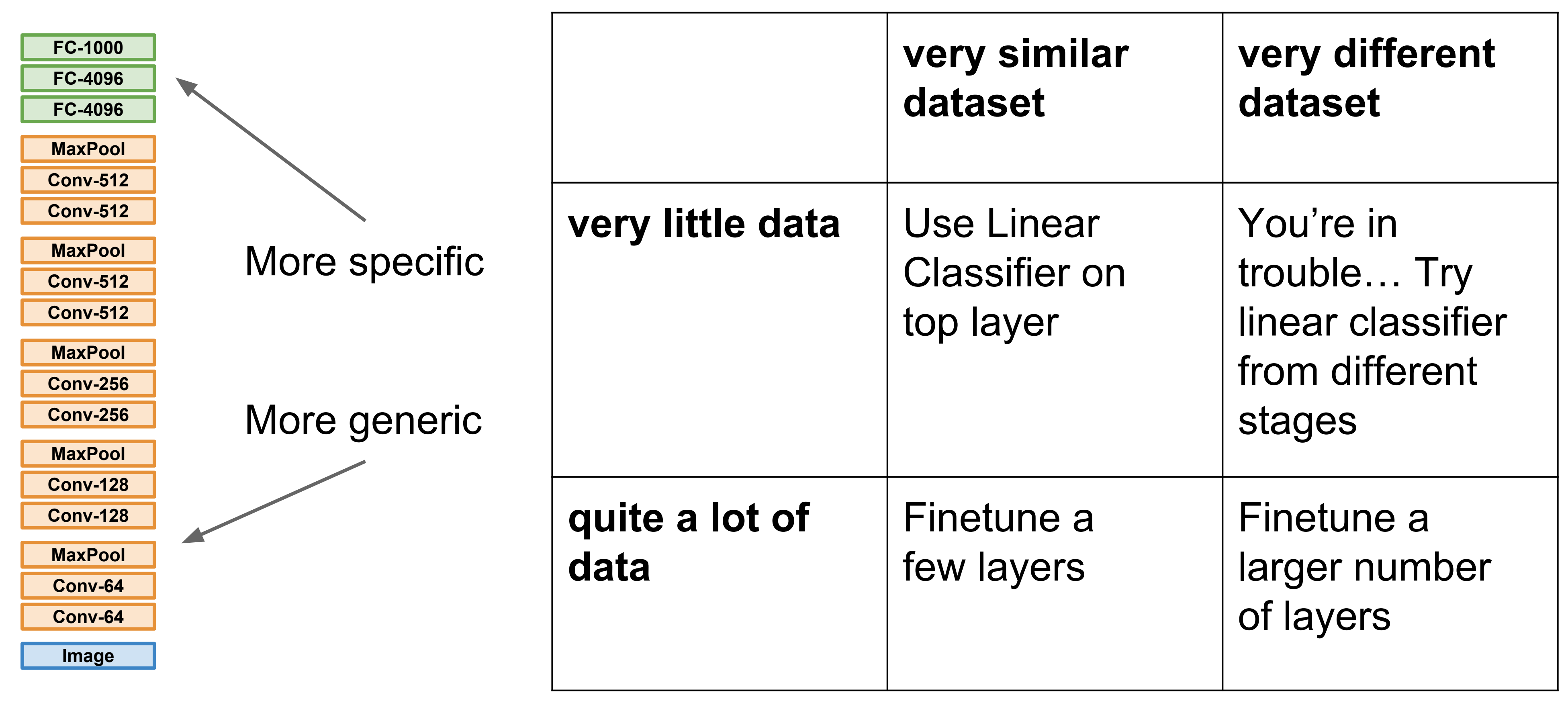
FC-C
FC-4096
FC-4096
MaxPool
Conv-512
Conv-512
MaxPool
Conv-512
Conv-512
MaxPool
Conv-256
Conv-256
MaxPool
Conv-128
Conv-128
MaxPool
Conv-64
Conv-64
Image

Train these

With bigger dataset, train more layers

Freeze these

Lower learning rate when finetuning; 1/10 of original LR is good starting point



More specific

More generic

	very similar dataset	**very different dataset**
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

Example Demo

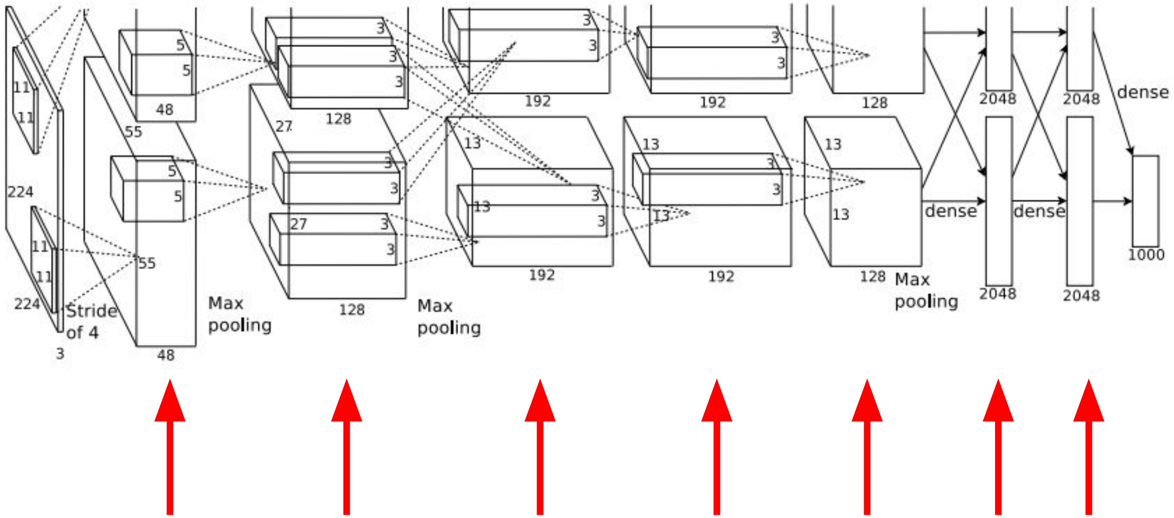
- Jupyter notebook with pytorch

Visualizing Convolutional Networks

Understanding intermediate neurons?

This image is CC0 public domain

Input Image:
3 x 224 x 224



Class Scores:
1000 numbers

What are the intermediate features looking for?

Visualizing CNN Features: Gradient Ascent

- **Gradient ascent:** Generate a synthetic image that maximally activates a neuron

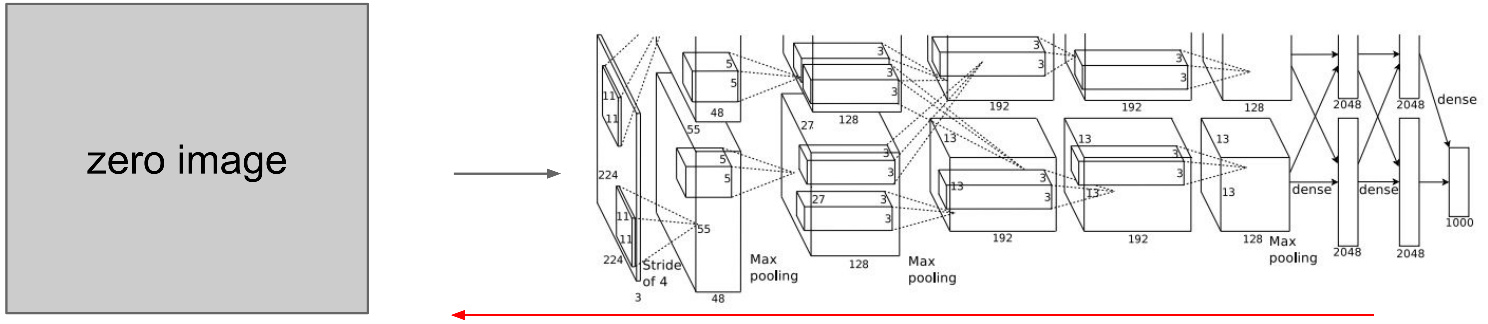
$$I^* = \arg\max_I \underbrace{\boxed{f(I)}}_{\text{Neuron value}} + \underbrace{\boxed{R(I)}}_{\text{Natural image regularizer}}$$

Visualizing CNN Features: Gradient Ascent

$$\arg\max_I \boxed{S_c(I)} - \lambda\|I\|_2^2$$

score for class c (before Softmax)

1. Initialize image to zeros



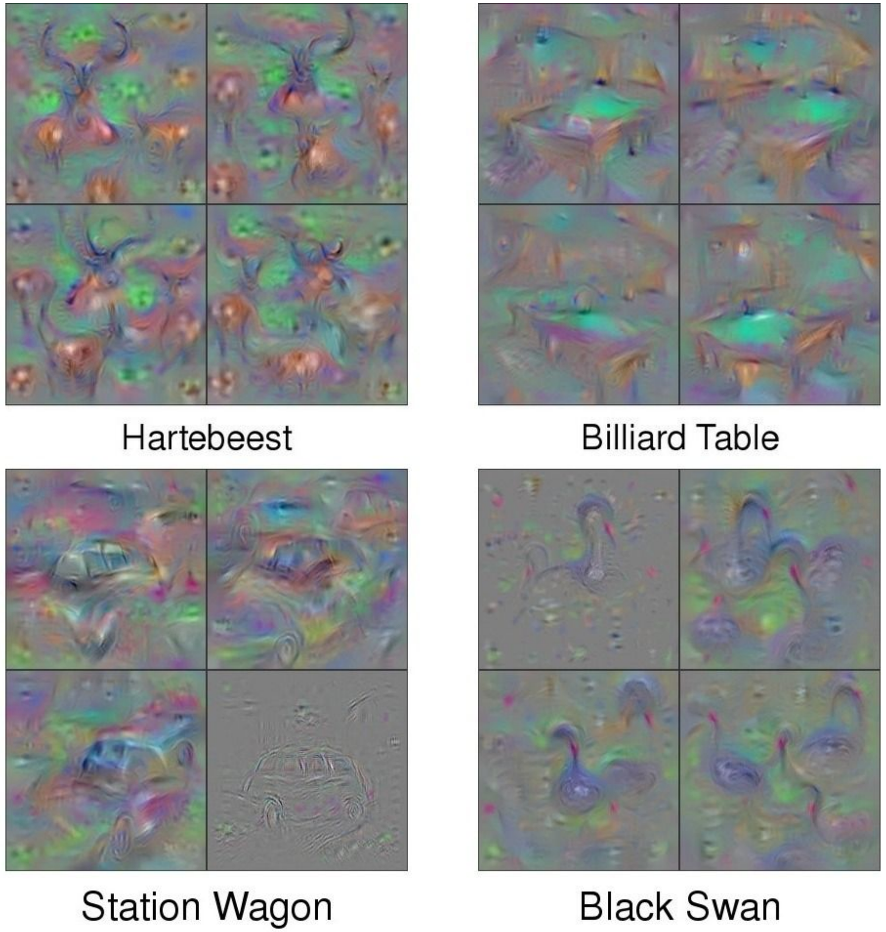
Repeat:
2. Forward image to compute current scores
3. Backprop to get gradient of neuron value with respect to image pixels
4. Make a small update to the image

Visualizing CNN Features: Gradient Ascent

$$\arg\max_I S_c(I) - \lambda\|I\|_2^2$$

Better regularizer: Penalize L2 norm of image; also during optimization periodically

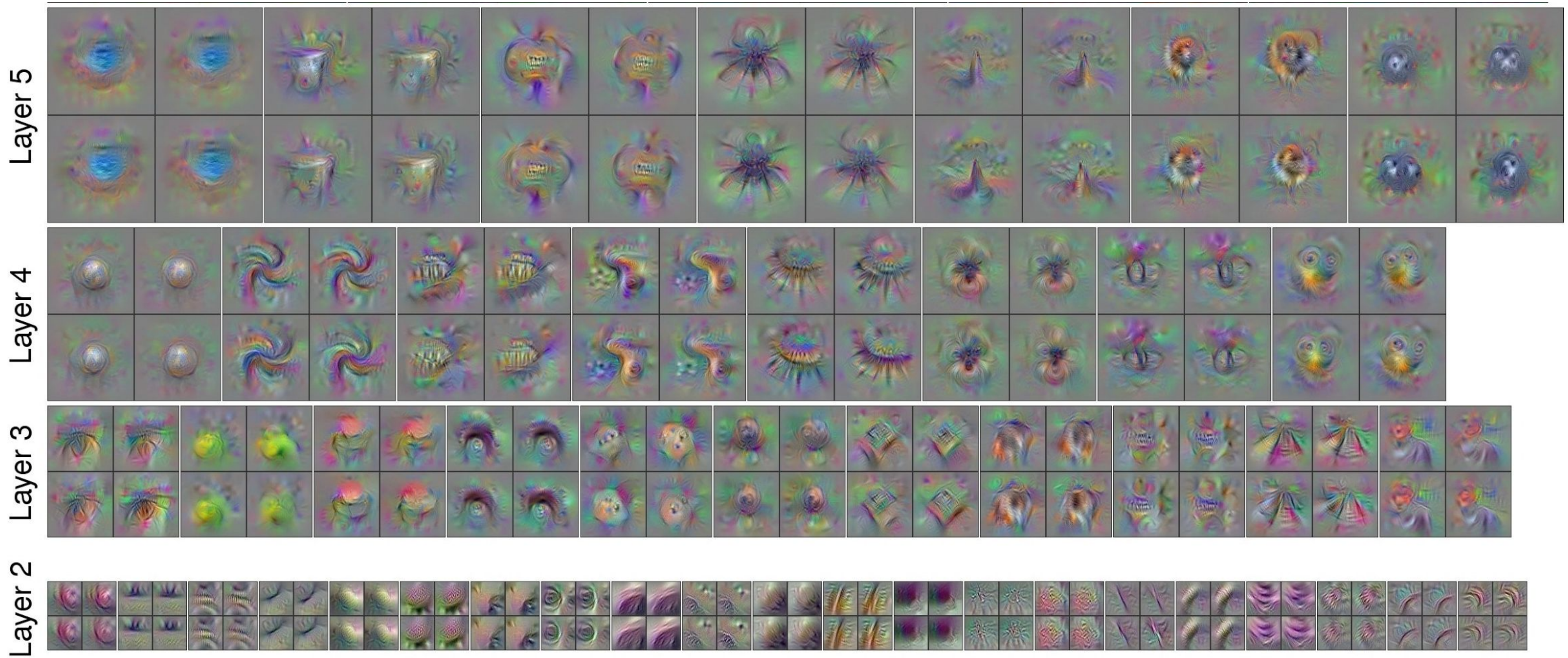
(1) Gaussian blur image
(2) Clip pixels with small values to 0
(3) Clip pixels with small gradients to 0



Yosinski et al, "Understanding Neural Networks Through Deep Visualization", ICML DL Workshop 2014.
Figure copyright Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson, 2014. Reproduced with permission.

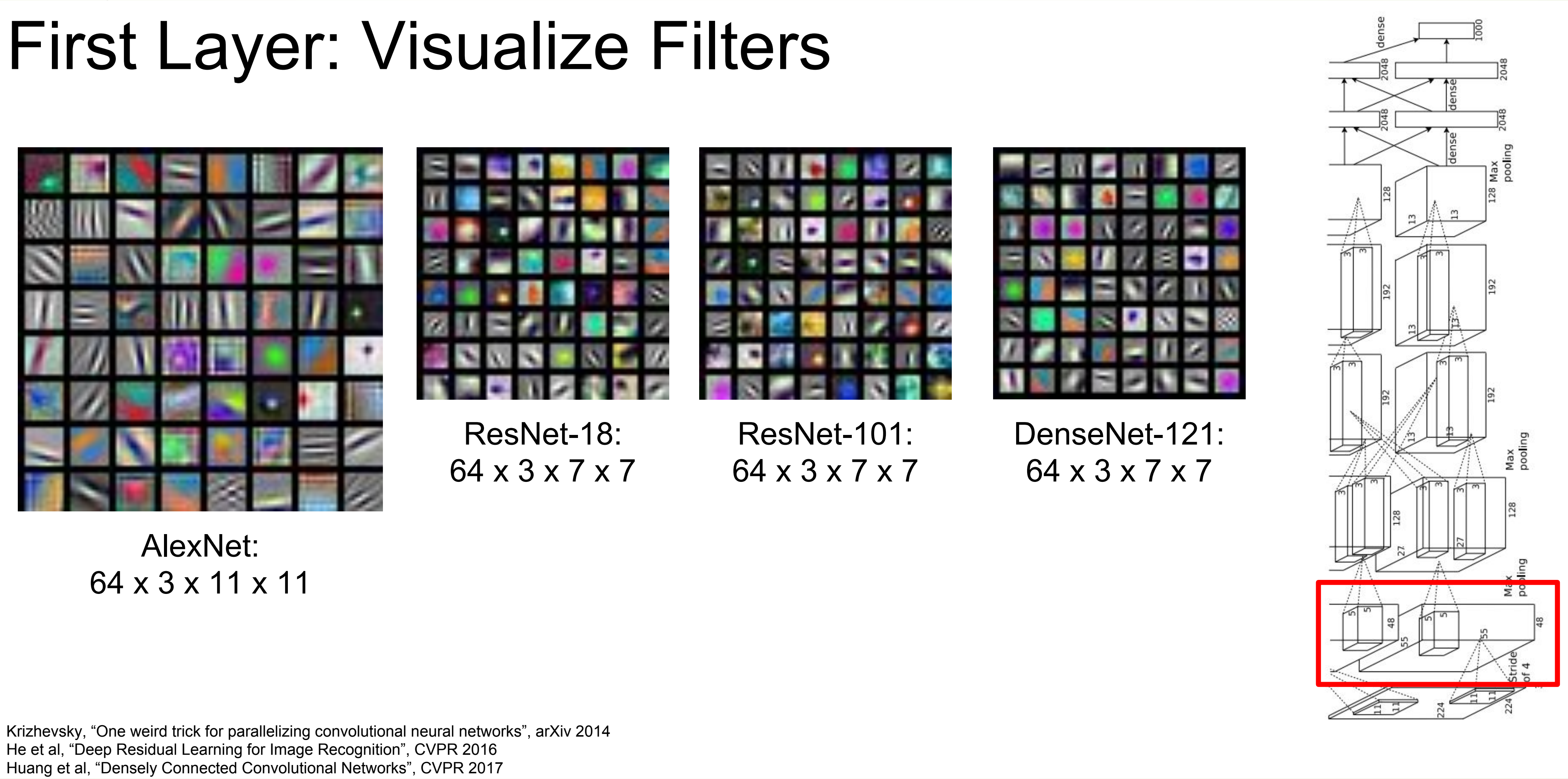
Visualizing CNN Features: Gradient Ascent

Use the same approach to visualize intermediate features



Yosinski et al, “Understanding Neural Networks Through Deep Visualization”, ICML DL Workshop 2014.
Figure copyright Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson, 2014. Reproduced with permission.

It’s easy to visualize early layers



First Layer: Visualize Filters

AlexNet:
64 x 3 x 11 x 11

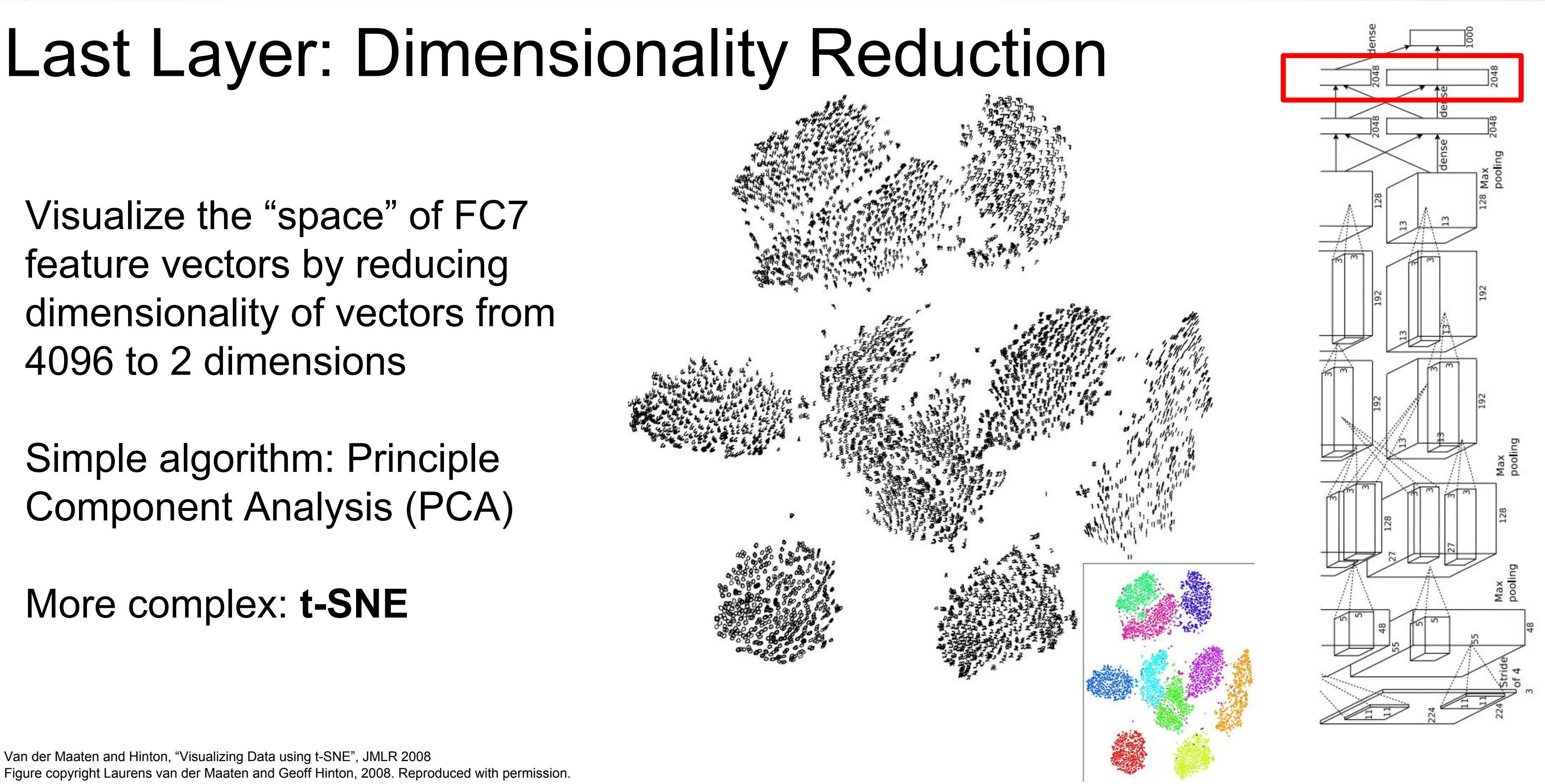
ResNet-18:
64 x 3 x 7 x 7

ResNet-101:
64 x 3 x 7 x 7

DenseNet-121:
64 x 3 x 7 x 7

Krizhevsky, “One weird trick for parallelizing convolutional neural networks”, arXiv 2014
He et al, “Deep Residual Learning for Image Recognition”, CVPR 2016
Huang et al, “Densely Connected Convolutional Networks”, CVPR 2017

Last layers are hard to visualize



Last Layer: Dimensionality Reduction

Visualize the “space” of FC7 feature vectors by reducing dimensionality of vectors from 4096 to 2 dimensions

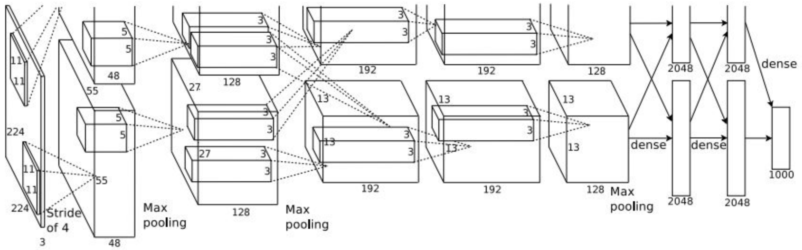
Simple algorithm: Principle Component Analysis (PCA)

More complex: **t-SNE**

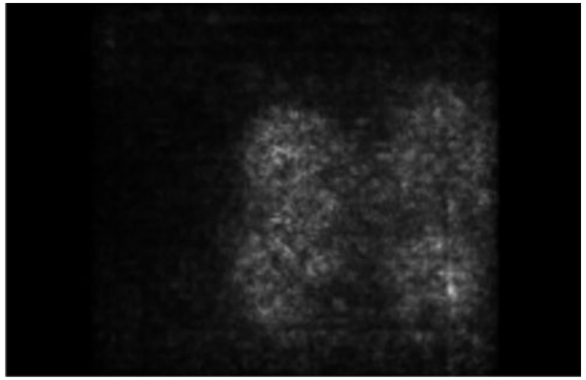
Van der Maaten and Hinton, “Visualizing Data using t-SNE”, JMLR 2008
Figure copyright Laurens van der Maaten and Geoff Hinton, 2008. Reproduced with permission.

Saliency Maps

How to tell which pixels matter for classification?



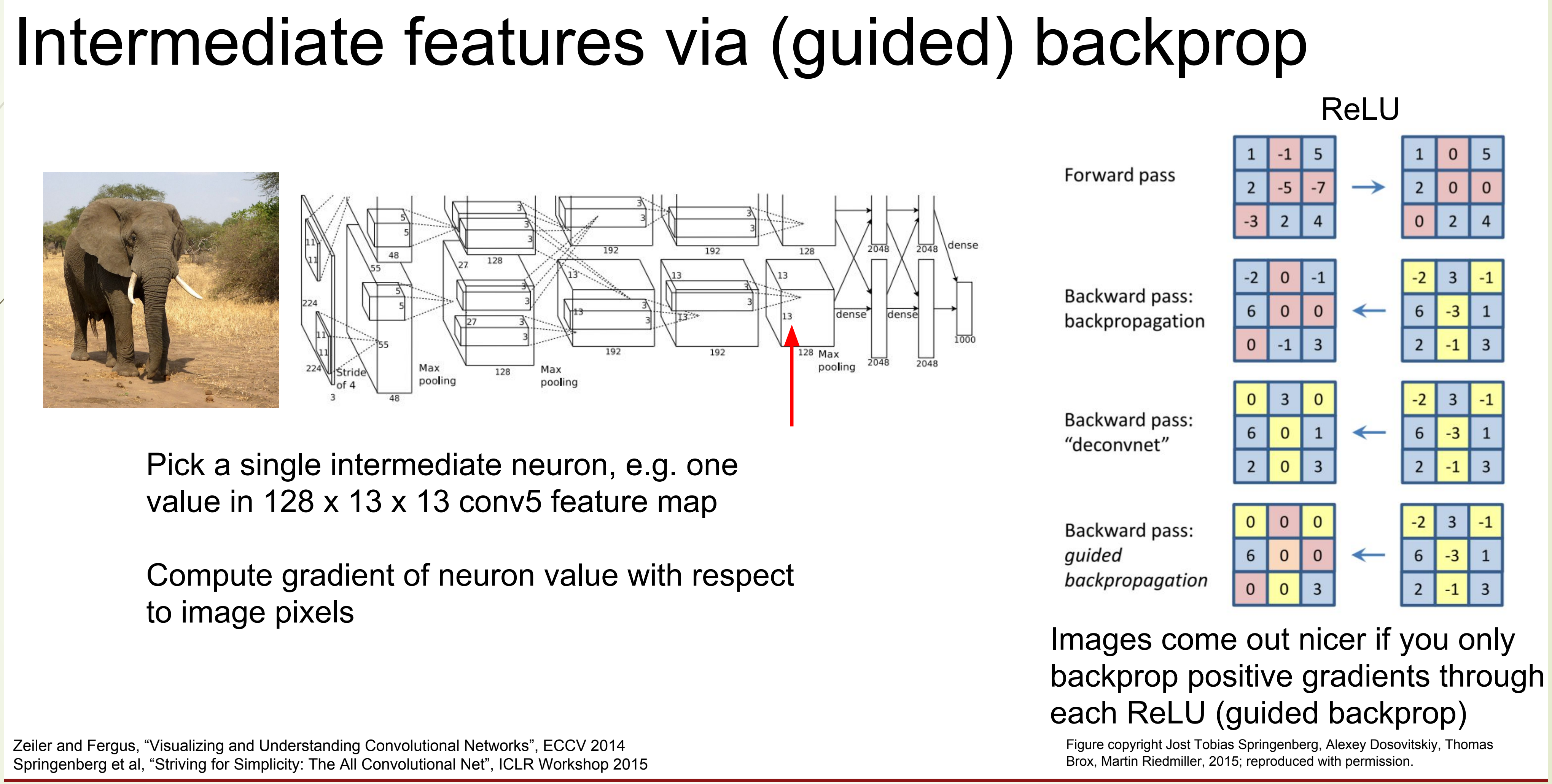
Dog



Compute gradient of (unnormalized) class score with respect to image pixels, take absolute value and max over RGB channels

Simonyan, Vedaldi, and Zisserman, “Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps”, ICLR Workshop 2014.

Guided BP



Intermediate features via (guided) backprop

Pick a single intermediate neuron, e.g. one value in 128 x 13 x 13 conv5 feature map

Compute gradient of neuron value with respect to image pixels

ReLU

Forward pass
$\begin{bmatrix}1 & -1 & 5\\ 2 & -5 & -7\\ -3 & 2 & 4\end{bmatrix} \rightarrow \begin{bmatrix}1 & 0 & 5\\ 2 & 0 & 0\\ 0 & 2 & 4\end{bmatrix}$

Backward pass: backpropagation
$\begin{bmatrix}-2 & 0 & -1\\ 6 & 0 & 0\\ 0 & -1 & 3\end{bmatrix} \leftarrow \begin{bmatrix}-2 & 3 & -1\\ 6 & -3 & 1\\ 2 & -1 & 3\end{bmatrix}$

Backward pass: "deconvnet"
$\begin{bmatrix}0 & 3 & 0\\ 6 & 0 & 1\\ 2 & 0 & 3\end{bmatrix} \leftarrow \begin{bmatrix}-2 & 3 & -1\\ 6 & -3 & 1\\ 2 & -1 & 3\end{bmatrix}$

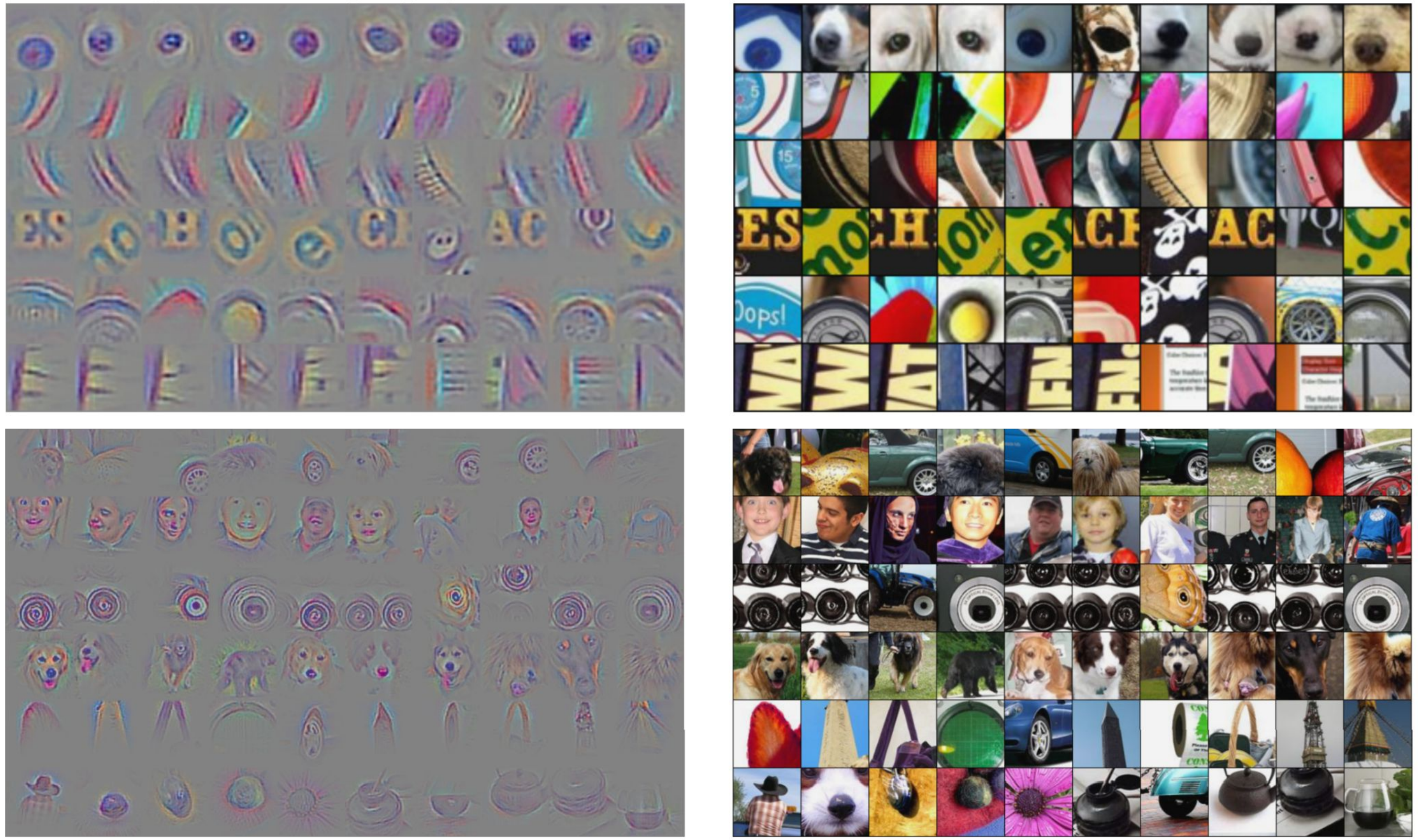
Backward pass: *guided backpropagation*
$\begin{bmatrix}0 & 0 & 0\\ 6 & 0 & 0\\ 0 & 0 & 3\end{bmatrix} \leftarrow \begin{bmatrix}-2 & 3 & -1\\ 6 & -3 & 1\\ 2 & -1 & 3\end{bmatrix}$

Images come out nicer if you only backprop positive gradients through each ReLU (guided backprop)

Zeiler and Fergus, "Visualizing and Understanding Convolutional Networks", ECCV 2014
Springenberg et al, "Striving for Simplicity: The All Convolutional Net", ICLR Workshop 2015

Figure copyright Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, Martin Riedmiller, 2015; reproduced with permission.

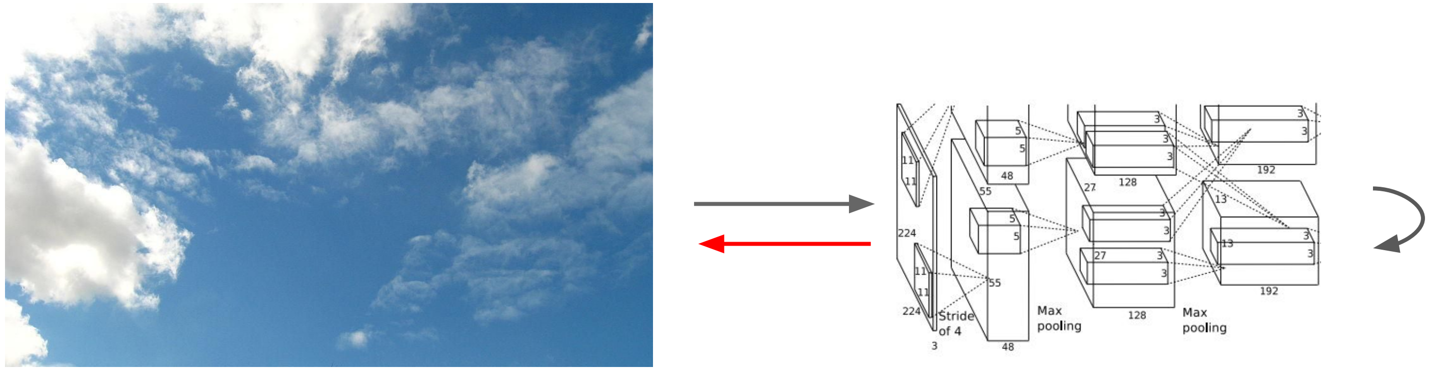
Intermediate features via Guided BP



Zeiler and Fergus, “Visualizing and Understanding Convolutional Networks”, ECCV 2014
Springenberg et al, “Striving for Simplicity: The All Convolutional Net”, ICLR Workshop 2015
Figure copyright Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, Martin Riedmiller, 2015; reproduced with permission.

DeepDream: amplifying features

Rather than synthesizing an image to maximize a specific neuron, instead try to **amplify** the neuron activations at some layer in the network



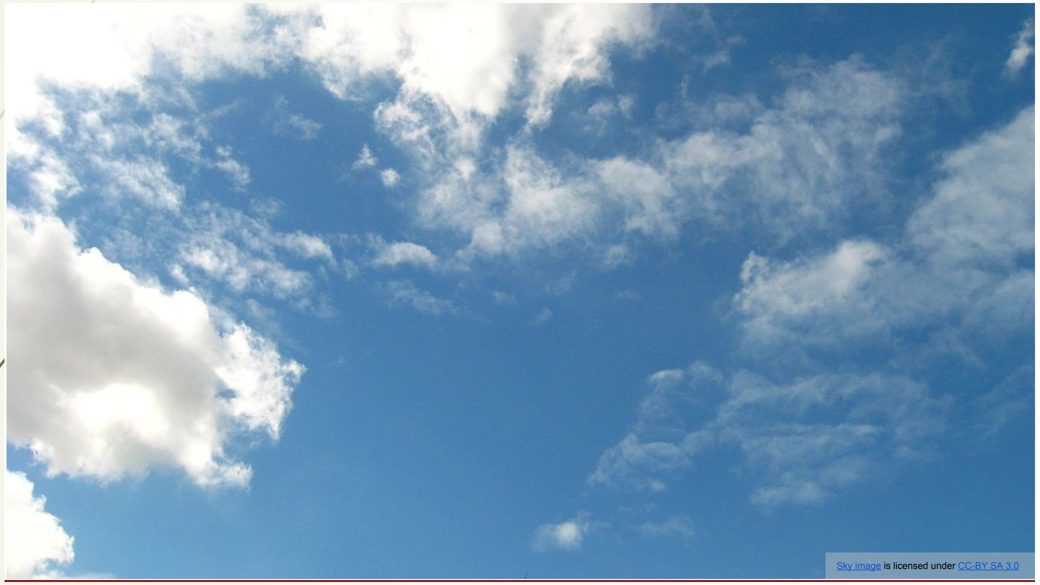
Choose an image and a layer in a CNN; repeat:

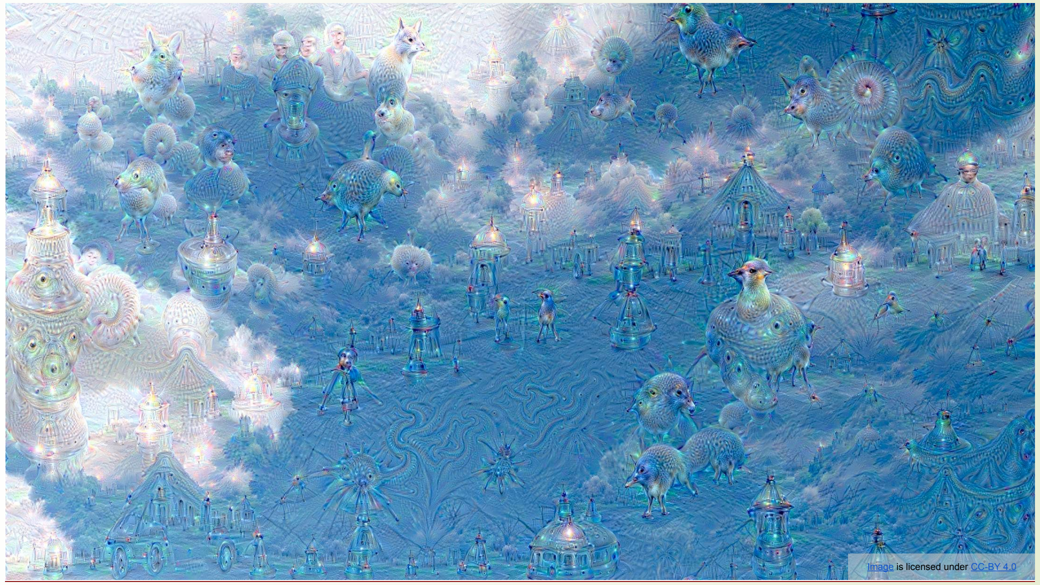
1. Forward: compute activations at chosen layer
2. Set gradient of chosen layer *equal to its activation*
3. Backward: Compute gradient on image
4. Update image

Equivalent to:

$$I^* = \arg\max_I \sum_i f_i(I)^2$$

Example: DeepDream of Sky






"Admiral Dog!"


"The Pig-Snail"


"The Camel-Bird"


"The Dog-Fish"

More Examples

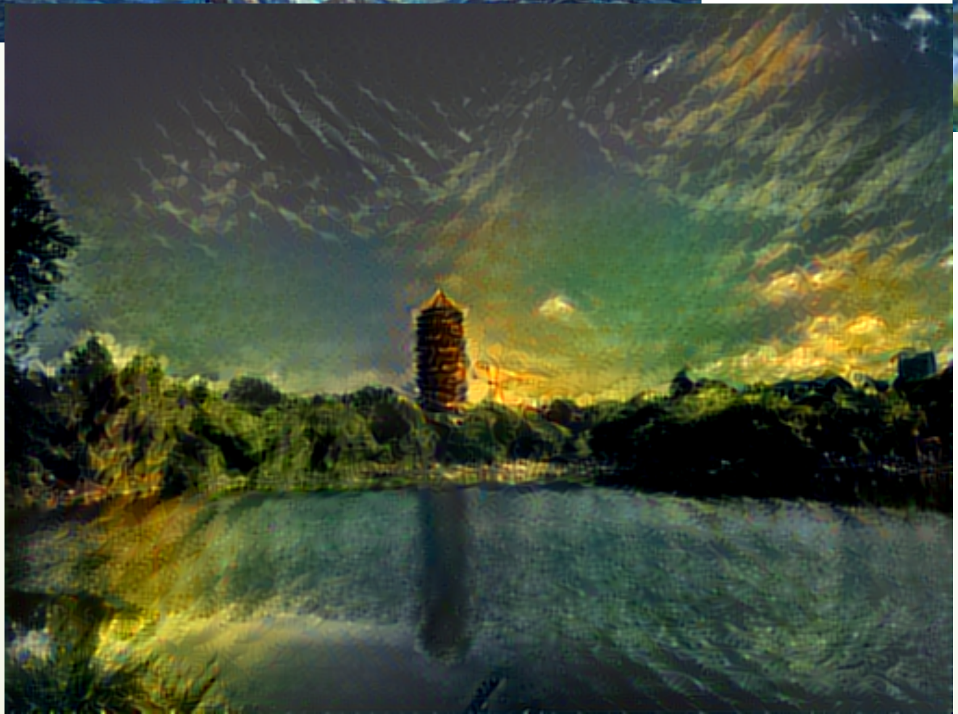
Python Notebooks

- An interesting Pytorch Implementation of these visualizatoin methods
  - https://github.com/utkuozbulak/pytorch-cnn-visualizations
- Some examples demo

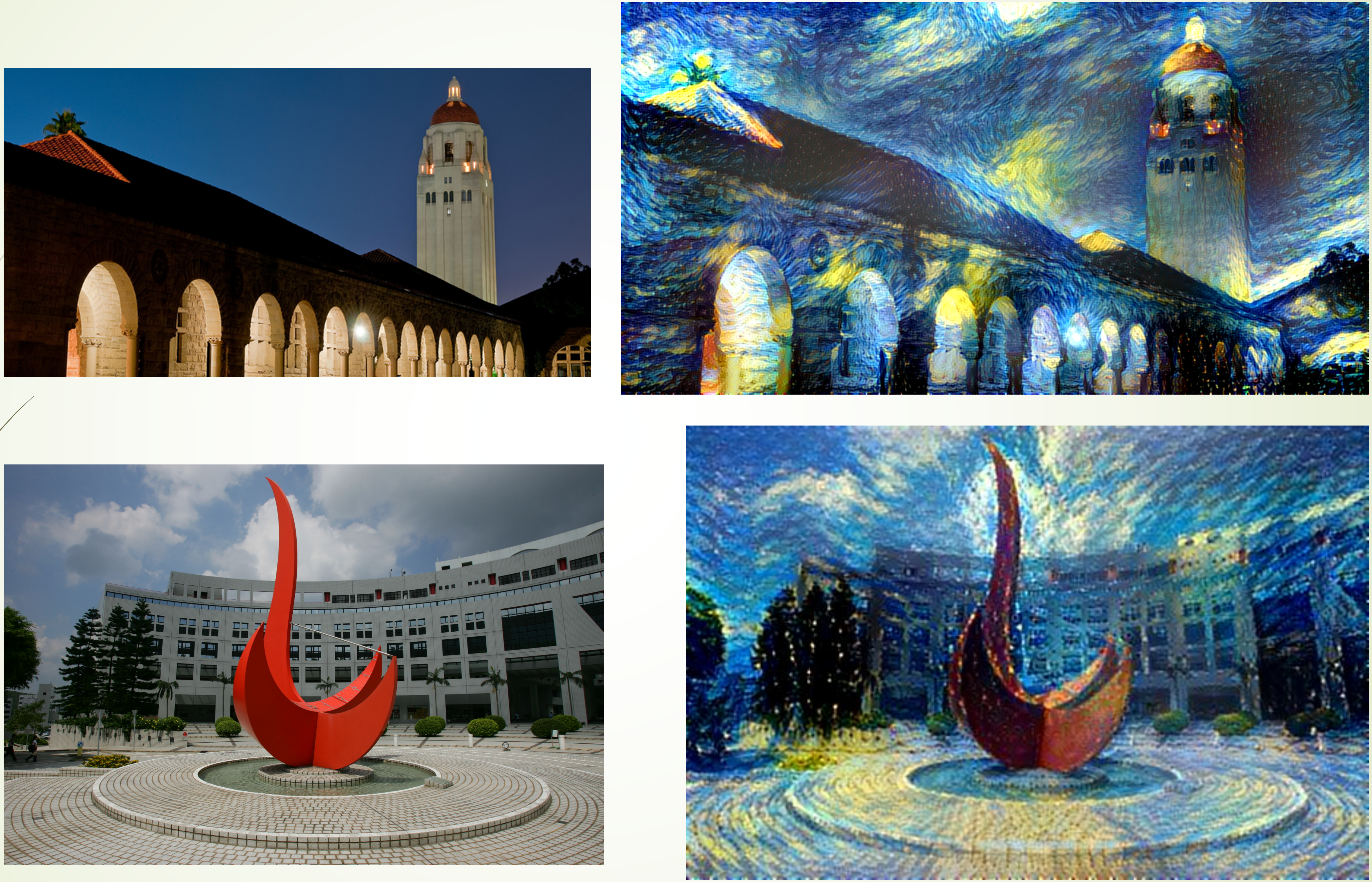
Neural Style

Example: The Noname Lake in PKU

Left: *Vincent Van Gogh*, Starry Night
Right: *Claude Monet*, Twilight Venice
Bottom: *William Turner*, Ship Wreck



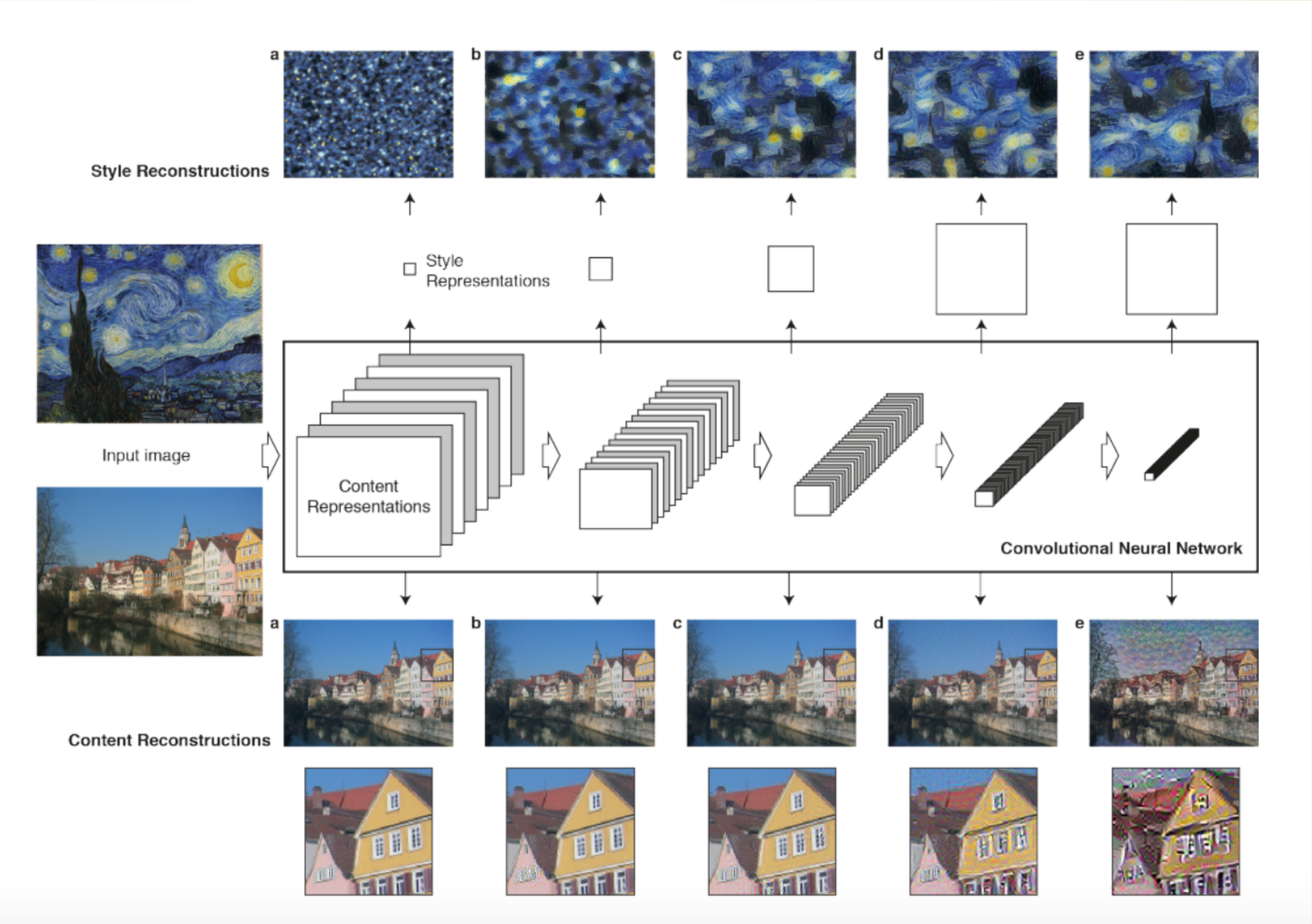
Application of Deep Learning:
Content-Style synthetic pictures
By “neural-style”



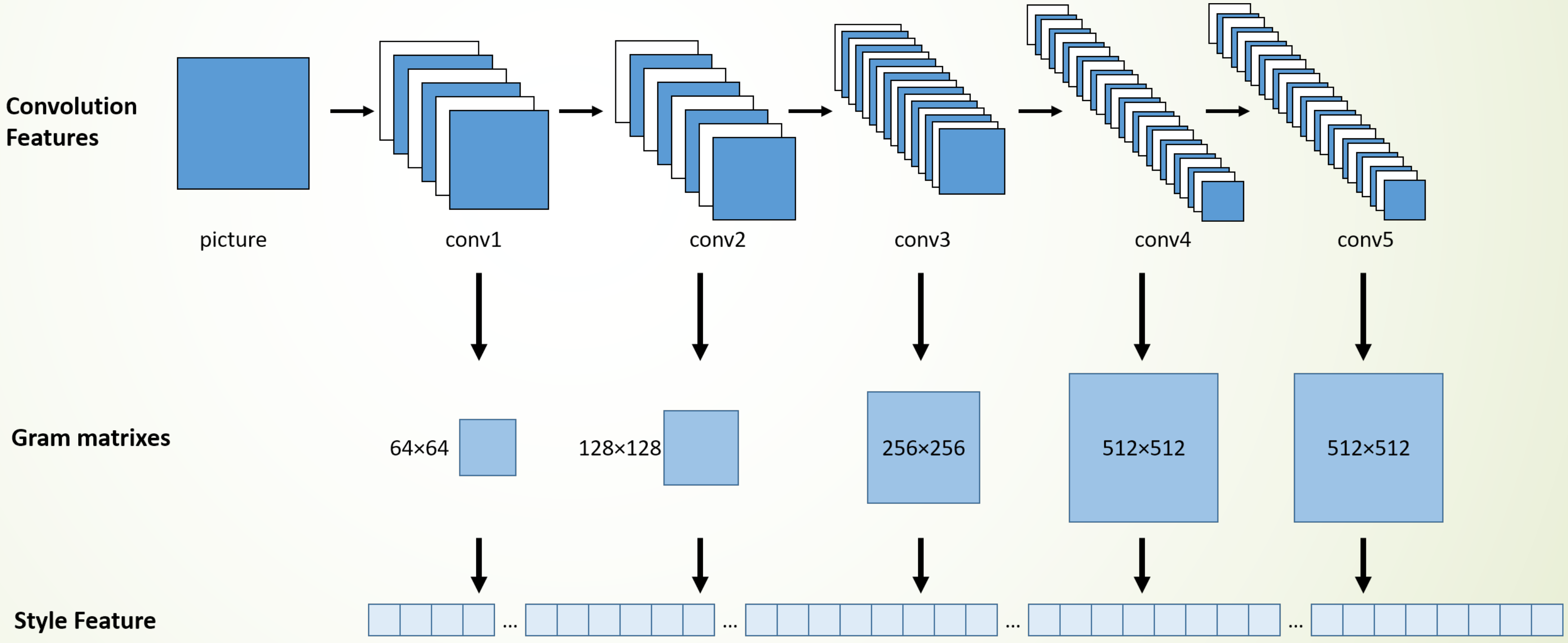
Neural Style

- J C Johnson’s Website: https://github.com/jcjohnson/neural-style
- A torch implementation of the paper
  - *A Neural Algorithm of Artistic Style*,
  - by Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge.
  - http://arxiv.org/abs/1508.06576

Style-Content Feature Extraction

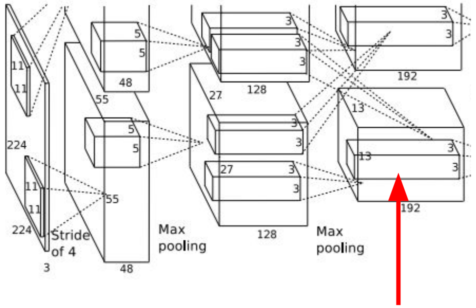


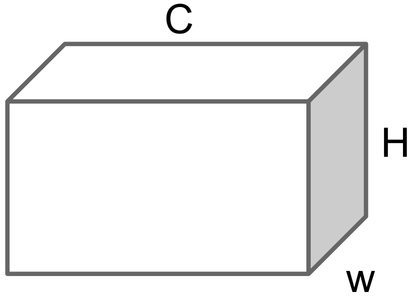
Style Features as Second Order Statistics

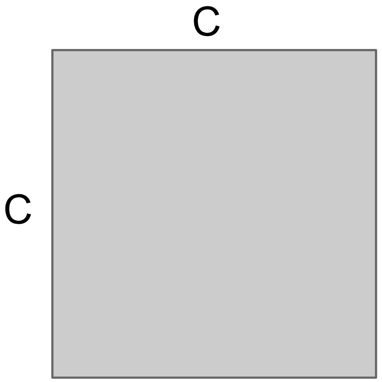


Neural Texture Synthesis

This image is in the public domain.







Each layer of CNN gives C x H x W tensor of features; H x W grid of C-dimensional vectors

Outer product of two C-dimensional vectors gives C x C matrix measuring co-occurrence

Average over all HW pairs of vectors, giving **Gram matrix** of shape C x C

Efficient to compute; reshape features from

C x H x W to =C x HW

then compute $G = FF^T$

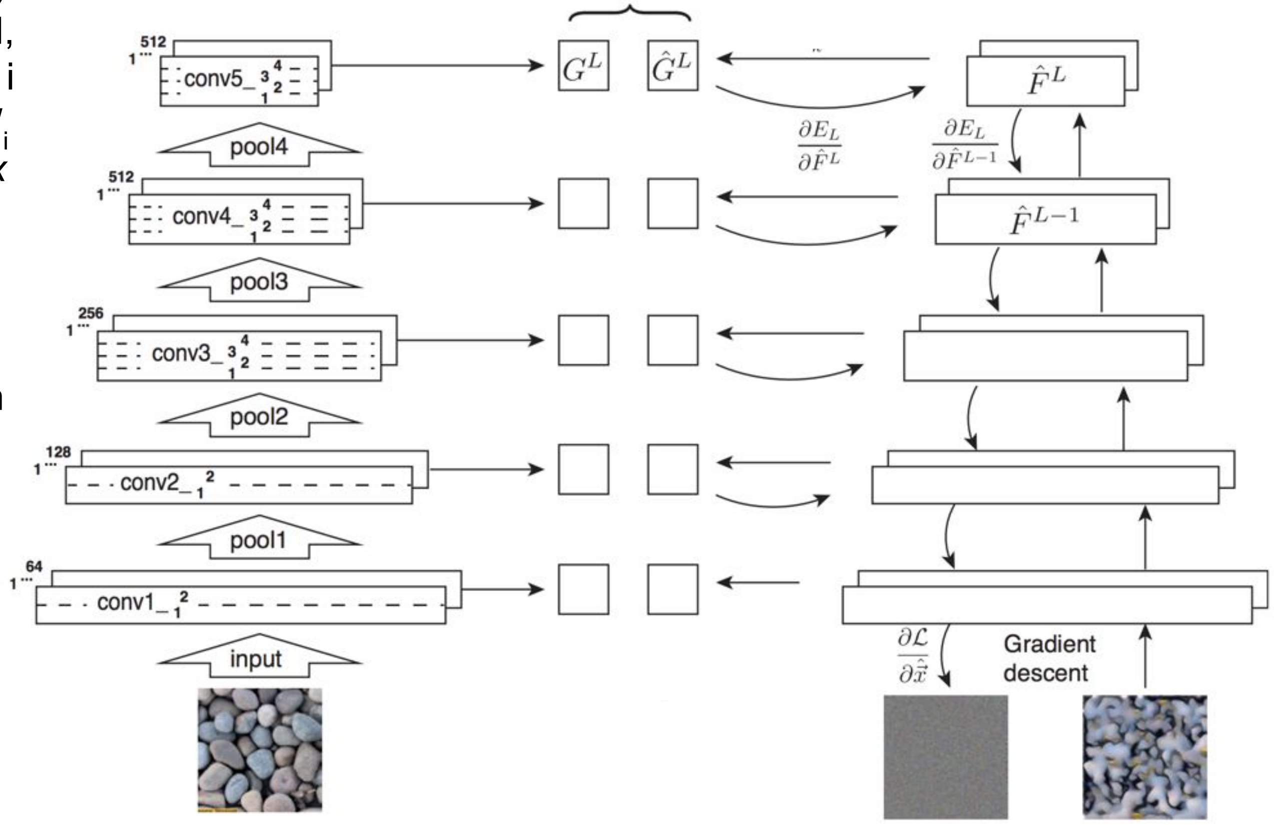
Neural Texture Synthesis

1. Pretrain a CNN on ImageNet (VGG-19)
2. Run input texture forward through CNN, record activations on every layer; layer i gives feature map of shape $C_i \times H_i \times W_i$
3. At each layer compute the *Gram matrix* giving outer product of features:

$$G_{ij}^{l} = \sum_{k} F_{ik}^{l} F_{jk}^{l} \quad \text{(shape } C_i \times C_i\text{)}$$

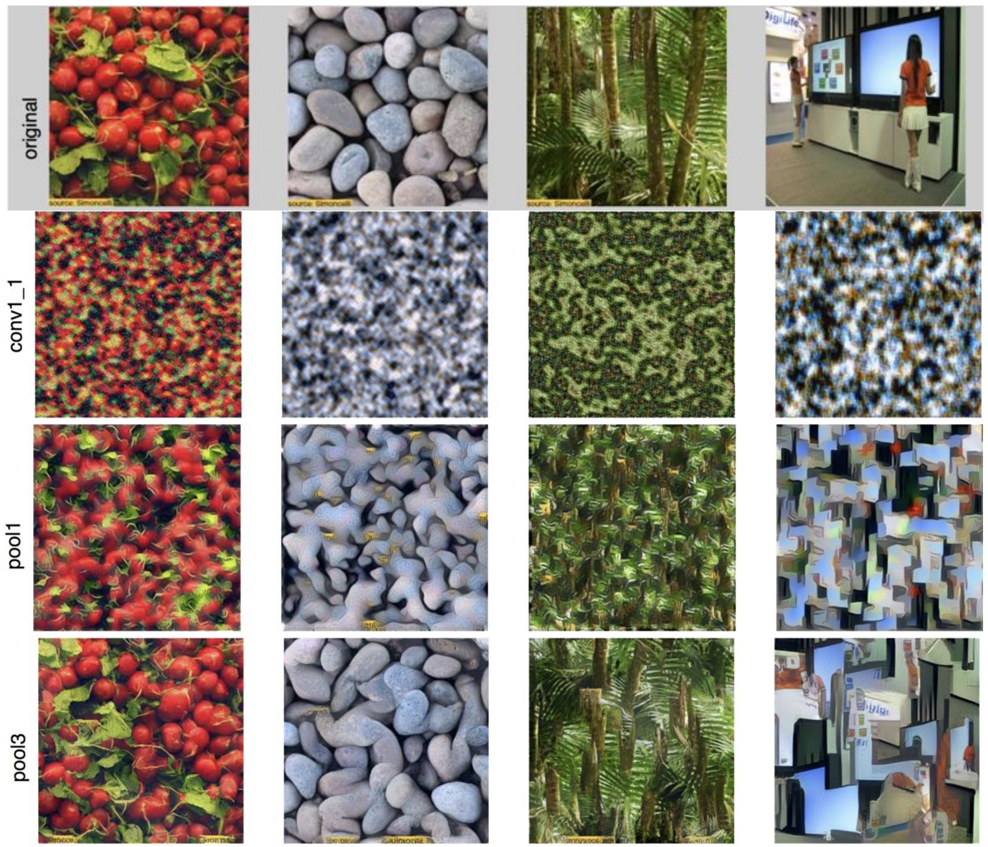
4. Initialize generated image from random noise
5. Pass generated image through CNN, compute Gram matrix on each layer
6. Compute loss: weighted sum of L2 distance between Gram matrices
7. Backprop to get gradient on image
8. Make gradient step on image
9. GOTO 5

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} \left(G_{ij}^{l} - \hat{G}_{ij}^{l} \right)^2 \qquad \mathcal{L}(\vec{x}, \hat{\vec{x}}) = \sum_{l=0}^{L} w_l E_l$$



Gatys, Ecker, and Bethge, "Texture Synthesis Using Convolutional Neural Networks", NIPS 2015
Figure copyright Leon Gatys, Alexander S. Ecker, and Matthias Bethge, 2015. Reproduced with permission.

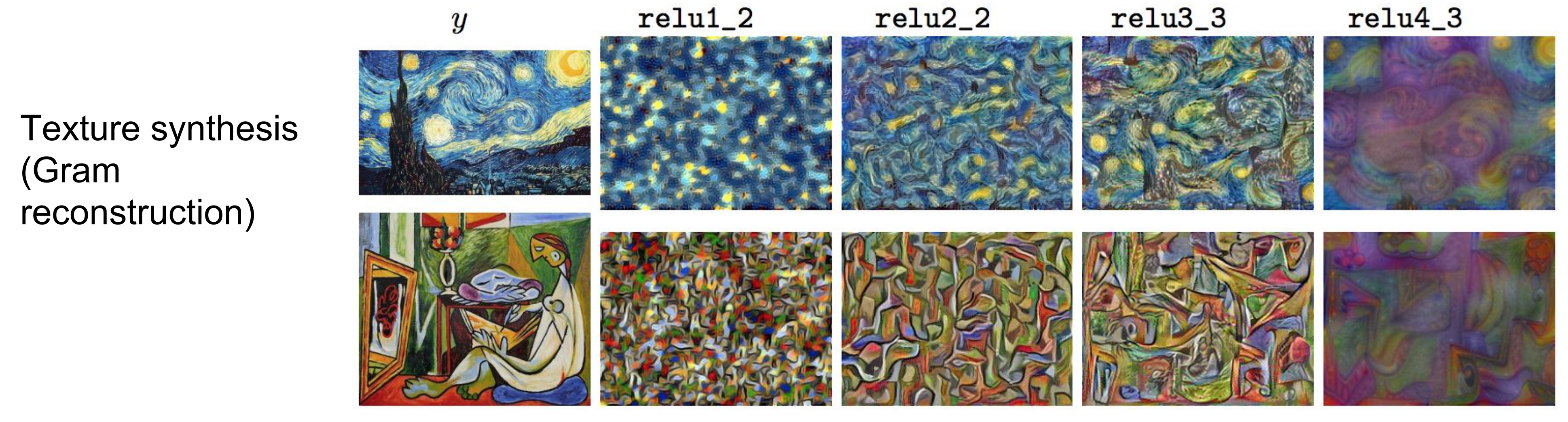
Neural Texture Synthesis



Reconstructing texture from higher layers recovers larger features from the input texture

Gatys, Ecker, and Bethge, “Texture Synthesis Using Convolutional Neural Networks”, NIPS 2015
Figure copyright Leon Gatys, Alexander S. Ecker, and Matthias Bethge, 2015. Reproduced with permission.

Neural Texture Synthesis: Gram Reconstruction


Texture synthesis (Gram reconstruction)

Feature Inversion

Given a CNN feature vector for an image, find a new image that:

- Matches the given feature vector
- “looks natural” (image prior regularization)

Given feature vector

$$\mathbf{x}^* = \underset{\mathbf{x}\in\mathbb{R}^{H\times W\times C}}{\operatorname{argmin}} \ \ell(\Phi(\mathbf{x}), \Phi_0) + \lambda\mathcal{R}(\mathbf{x})$$

Features of new image

$$\ell(\Phi(\mathbf{x}), \Phi_0) = \|\Phi(\mathbf{x}) - \Phi_0\|^2$$

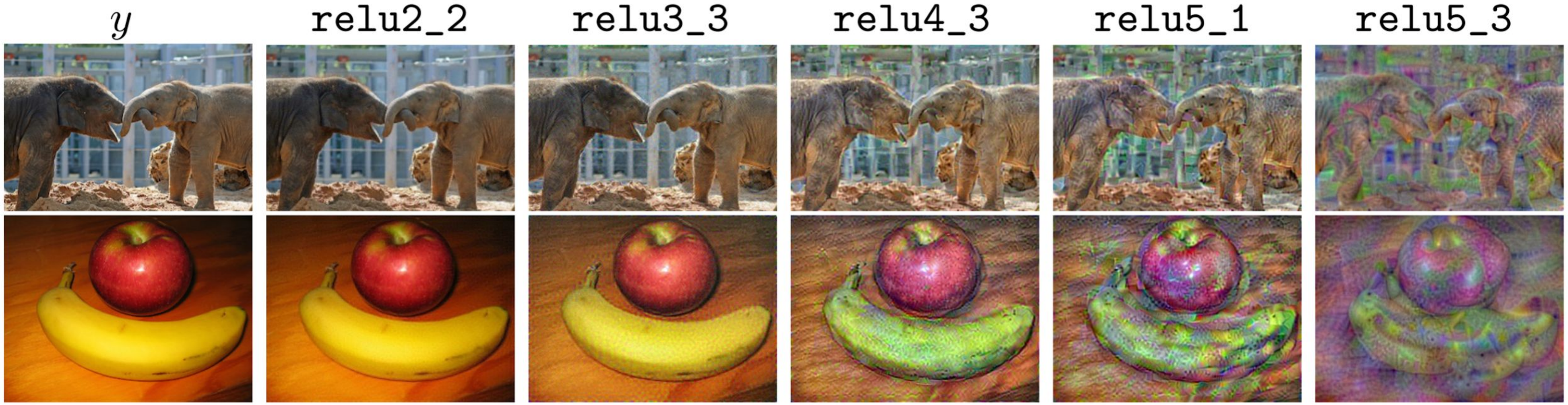
$$\mathcal{R}_{V^\beta}(\mathbf{x}) = \sum_{i,j}\left(\left(x_{i,j+1} - x_{ij}\right)^2 + \left(x_{i+1,j} - x_{ij}\right)^2\right)^{\frac{\beta}{2}}$$

Total Variation regularizer
(encourages spatial smoothness)

Mahendran and Vedaldi, “Understanding Deep Image Representations by Inverting Them”, CVPR 2015

Feature Inversion

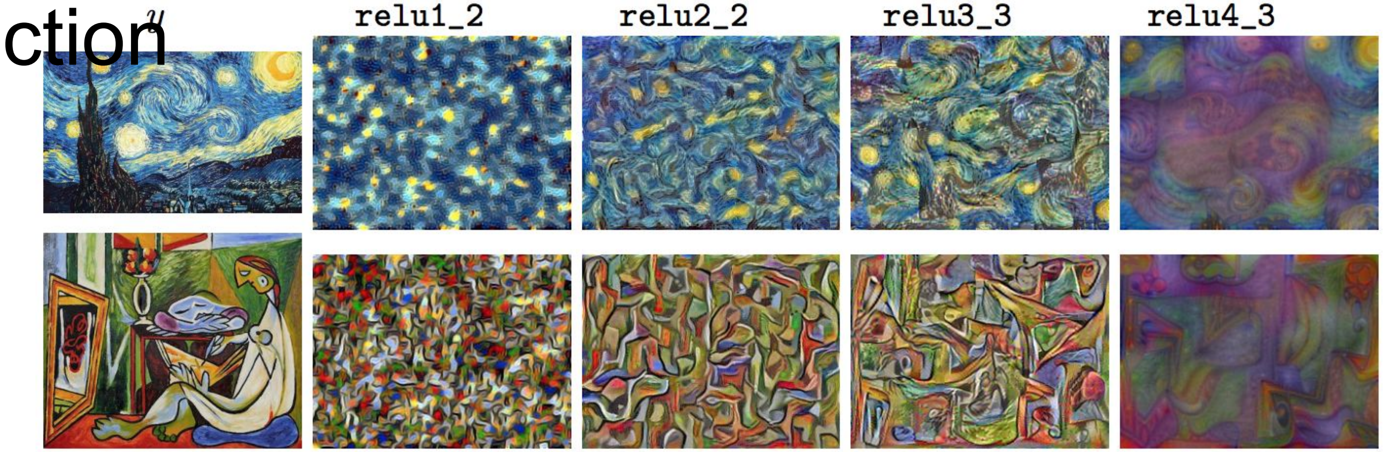
Reconstructing from different layers of VGG-16



Mahendran and Vedaldi, “Understanding Deep Image Representations by Inverting Them”, CVPR 2015
Figure from Johnson, Alahi, and Fei-Fei, “Perceptual Losses for Real-Time Style Transfer and Super-Resolution”, ECCV 2016. Copyright Springer, 2016.
Reproduced for educational purposes.

Neural Style Transfer: Feature + Gram Reconstruction

Texture synthesis (Gram reconstruction)



Feature reconstruction

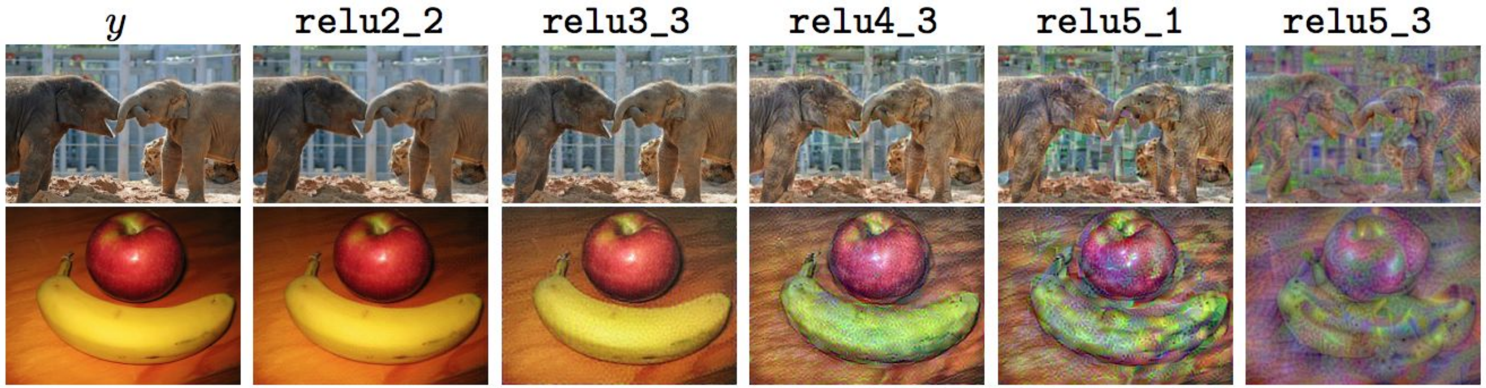


Figure from Johnson, Alahi, and Fei-Fei, “Perceptual Losses for Real-Time Style Transfer and Super-Resolution”, ECCV 2016. Copyright Springer, 2016. Reproduced for educational purposes.

Combined Loss for both Content (1st order statistics) and Style (2nd order statistics: Gram)

$$\mathcal{L}_{content}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum_{i,j} \left(F^l_{ij} - P^l_{ij}\right)^2 .$$

$$\mathcal{L}_{style}(\vec{a}, \vec{x}) = \sum_{l=0}^{L} w_l E_l$$

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} \left(G^l_{ij} - A^l_{ij}\right)^2$$

$$G^l_{ij} = \sum_k F^l_{ik} F^l_{jk}.$$

Neural Style Transfer

Content Image

\+

Style Image

=

Style Transfer!

This image is licensed under CC-BY 3.0

Starry Night by Van Gogh is in the public domain

This image copyright Justin Johnson, 2015. Reproduced with permission.

CNN learns texture features, not shapes!

(a) Texture image

81.4%	**Indian elephant**
10.3%	indri
8.2%	black swan

(b) Content image

71.1%	**tabby cat**
17.3%	grey fox
3.3%	Siamese cat

(c) Texture-shape cue conflict

63.9%	**Indian elephant**
26.4%	indri
9.6%	black swan

Geirhos et al. ICLR 2019

https://videoken.com/embed/W2HvLBMhCJQ?tocitem=46

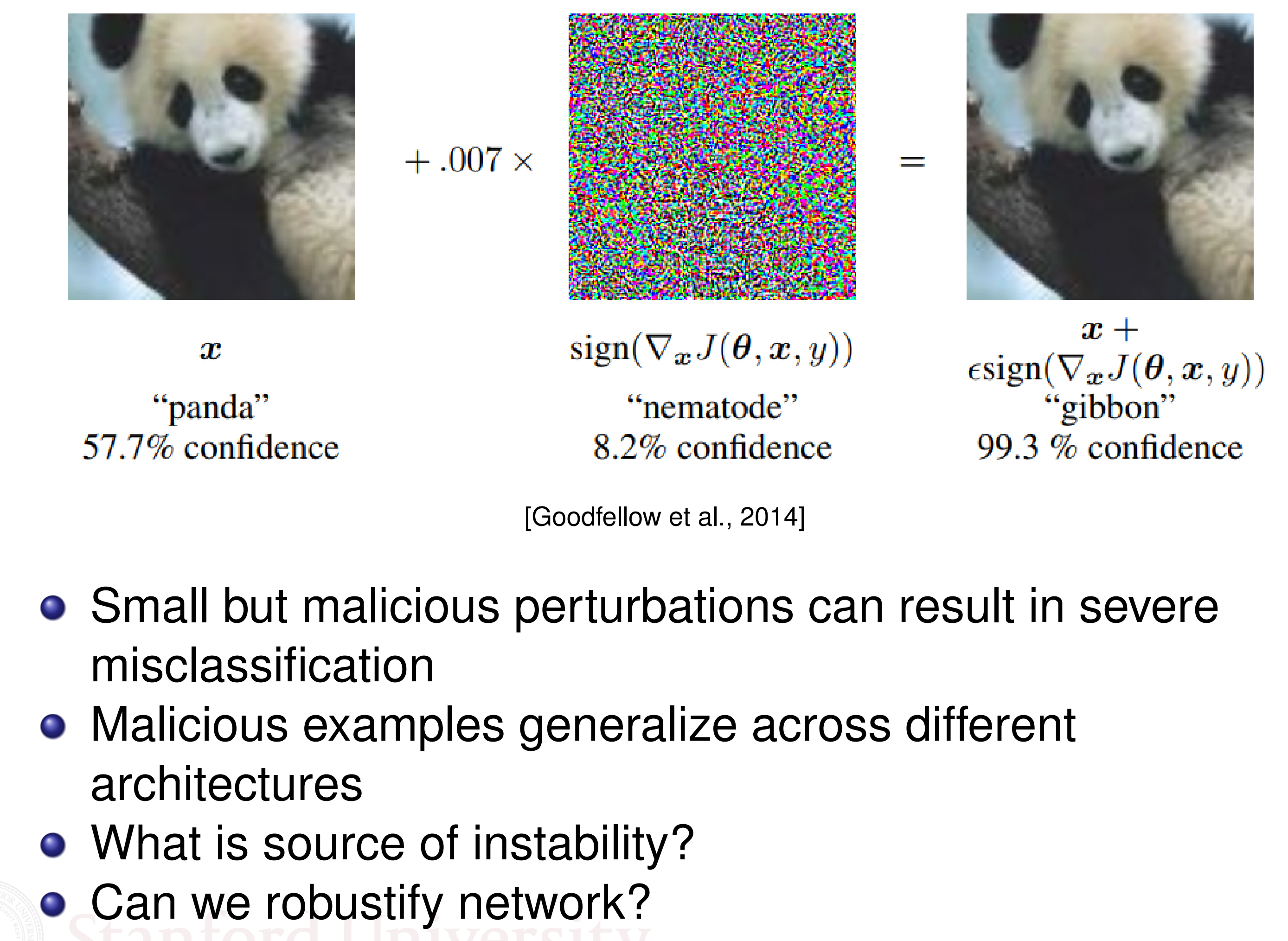
1:16:47

Examples

- Jupyter Notebook Demo

Adversarial Examples and Robustness

Deep Learning may be fragile: adversarial examples



x + .007 × $\text{sign}(\nabla_{x} J(\theta, x, y))$ = $x + \epsilon\,\text{sign}(\nabla_{x} J(\theta, x, y))$

"panda"
57.7% confidence

"nematode"
8.2% confidence

"gibbon"
99.3 % confidence

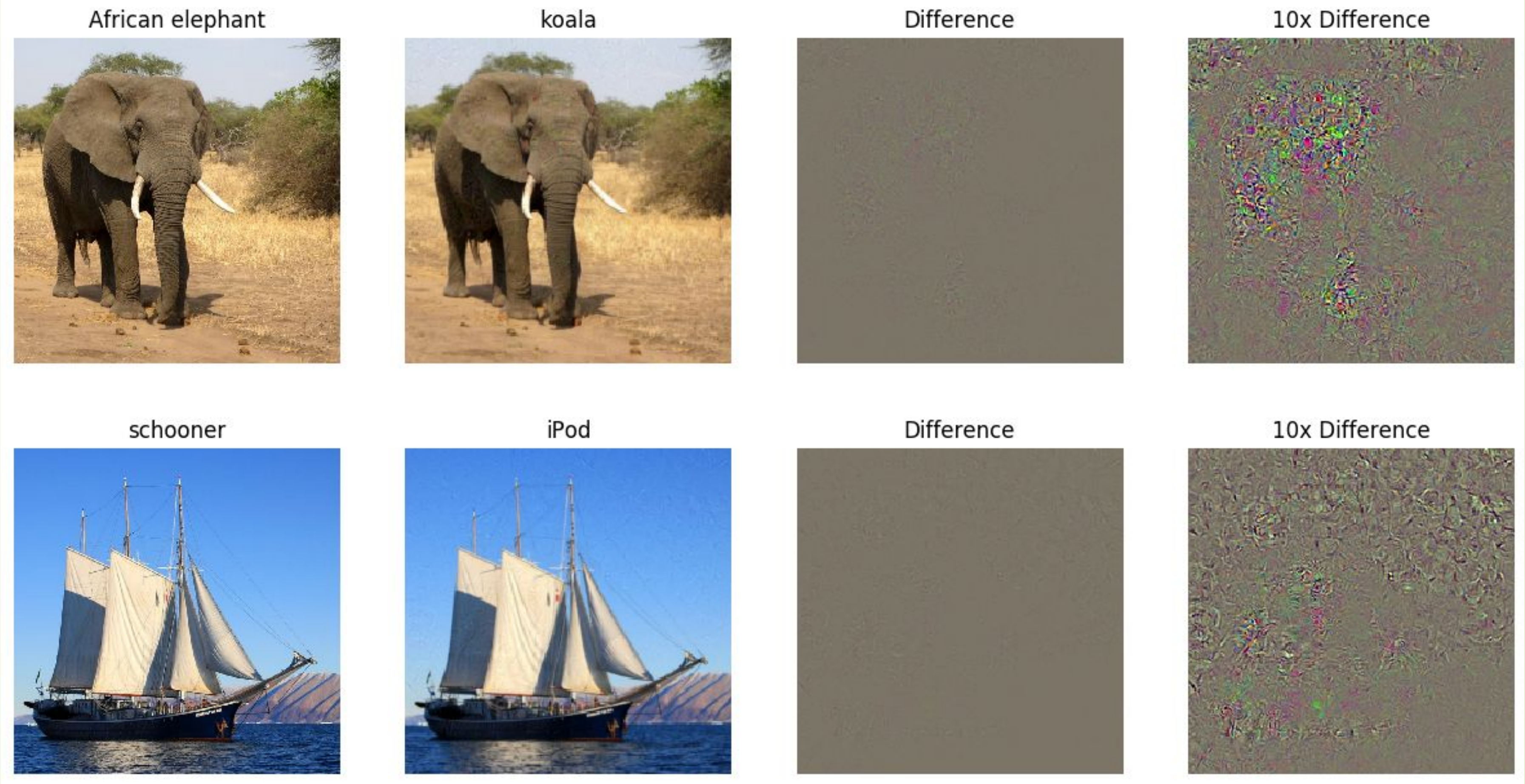
[Goodfellow et al., 2014]

- Small but malicious perturbations can result in severe misclassification
- Malicious examples generalize across different architectures
- What is source of instability?
- Can we robustify network?

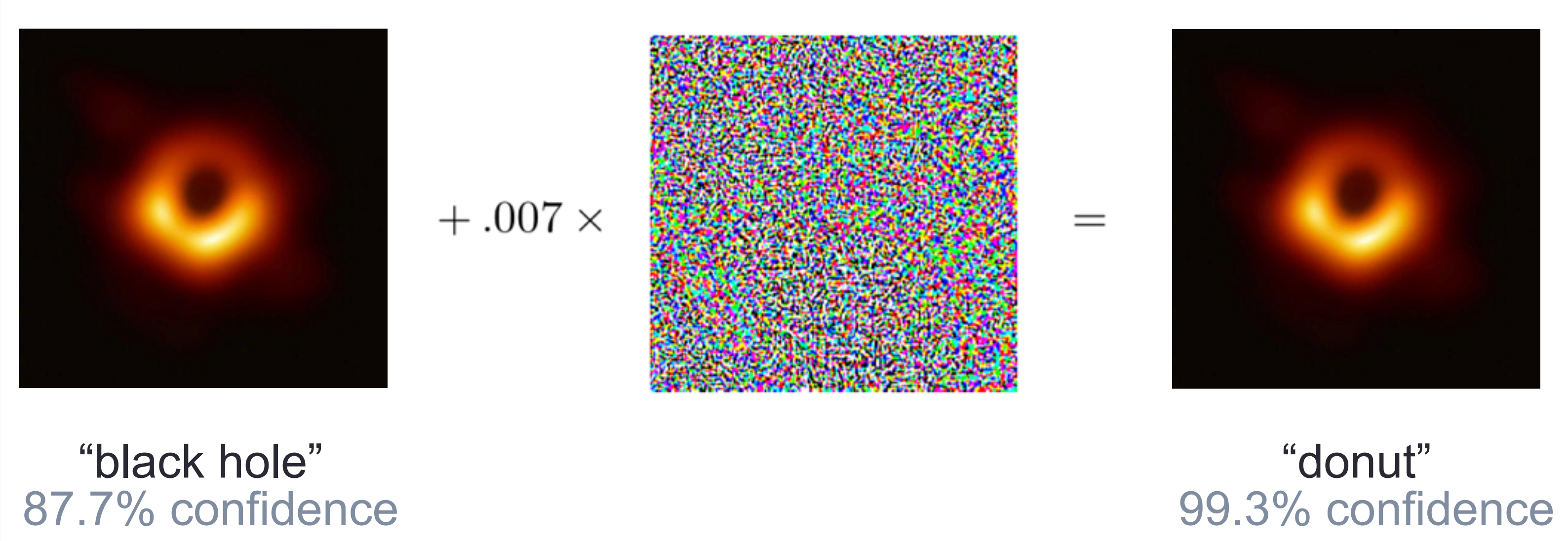
Adversarial Examples: Fooling Images

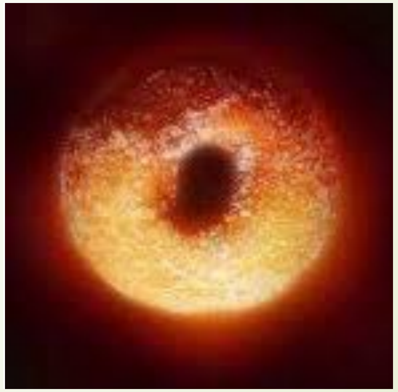
- Start from an arbitrary image
- Pick an arbitrary class
- Modify the image to maximize the class
- Repeat until network is fooled

Fooling Images/Adversarial Examples



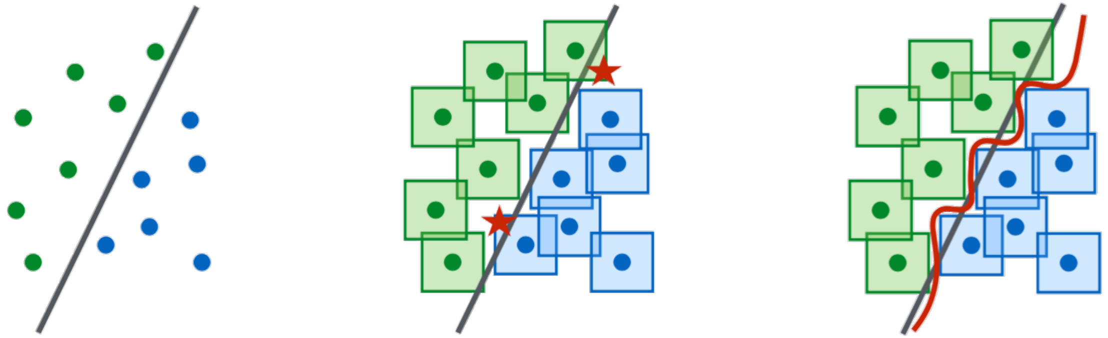
Convolutional Networks lack Robustness





Courtesy of Dr. Hongyang ZHANG.

Adversarial Robust Training



- Traditional training:

$$\min_{\theta} J_n(\theta, \mathbf{z} = (x_i, y_i)_{i=1}^{n})$$

  - e.g. square or cross-entropy loss as negative log-likelihood of logit models
- Robust optimization (Madry et al. ICLR'2018):

$$\min_{\theta} \max_{\|\epsilon_i\| \le \delta} J_n(\theta, \mathbf{z} = (x_i + \epsilon_i, y_i)_{i=1}^{n})$$

  - robust to any distributions, yet computationally hard

Extended by **Hongyang ZHANG** et al. by TRADES, 2019.

Thank you!